Course Title:

Compiler Construction

Assignment Title:

Mini Optimization Compiler

Submitted to:

Sir Safdar

Submitted by:

(Yasir Fareed, Umer Qazi, Huma Ibrar)

Registration No:

(22245, 22068 , 22292)

Program:

BSCS- 6th Semester

Abstract :

This project implements a Mini Optimizing Compiler (MOC) that translates a subset of C++ programs into tokens, Three Address Code (TAC), assembly like instructions, and final output with comprehensive error reporting. The compiler performs all major phases of compilation, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. It supports basic data types, arithmetic operations, control flow structures, and input/output operations using cout, with a strong emphasis on code optimization techniques including constant folding and dead code elimination.

1. Introduction :

A compiler is a software tool that translates source code written in a high-level programming language into machine code or intermediate code that a computer can execute. The Mini Optimizing Compiler (MOC) focuses on implementing the core phases of compilation for a selected subset of the C++ language with particular emphasis on optimization techniques.

2. Objectives :

- Implement lexical analysis using regular expressions to tokenize the input source code.
- Perform syntax checking to ensure grammatical correctness of C++ subset programs.
- Conduct semantic analysis for type checking, symbol validation, and type inference.
- Generate Three Address Code (TAC) as an intermediate representation.
- Implement optimization techniques including constant folding and dead code elimination.
- Translate optimized TAC into assembly-like instructions.
- Provide comprehensive error handling across all compiler phases.
- Execute the generated code using a virtual machine interpreter.

How to run the Project :

Prerequisites

- Python 3.12 or higher installed on your system
- VS Code with Live Server extension installed
- Basic familiarity with command line interface

Setup and Execution Steps

Step 1 : Clone/Download the Project

If you haven't already, download the project repository to your local machine and navigate to the project directory :

cd path/to/CompilerProject

Step 2 : Set Up Dependencies

1. Navigate to the backend folder:

bash command :

```
cd backend
```

2. Install the required Python packages (Flask and Flask-CORS):

bash command :

```
pip install flask flask-cors
```

Step 3 : Run the Backend Server

1. From the backend folder, execute the following command:

bash command :

```
python3 app.py
```

2. You should see output indicating that the Flask server is running on `http://localhost:5000`

Step 4 : Launch the Frontend

1. Open the project in VS Code:

bash command :

```
code ..
```

2. In VS Code, navigate to the **`index.html`** file in the root directory

3. Right-click on the **`index.html`** file and select **"Open with Live Server"**

(Ensure you have the Live Server extension installed in VS Code)

4. This will automatically open the web application in your default browser at **`http://127.0.0.1:5500`**

Step 5 : Using the Compiler

- The web interface will load with a code editor on the left panel

- Enter your C++ subset code in the editor

- Click the "Compile" button to process your code

- The results will appear in the right panel showing:

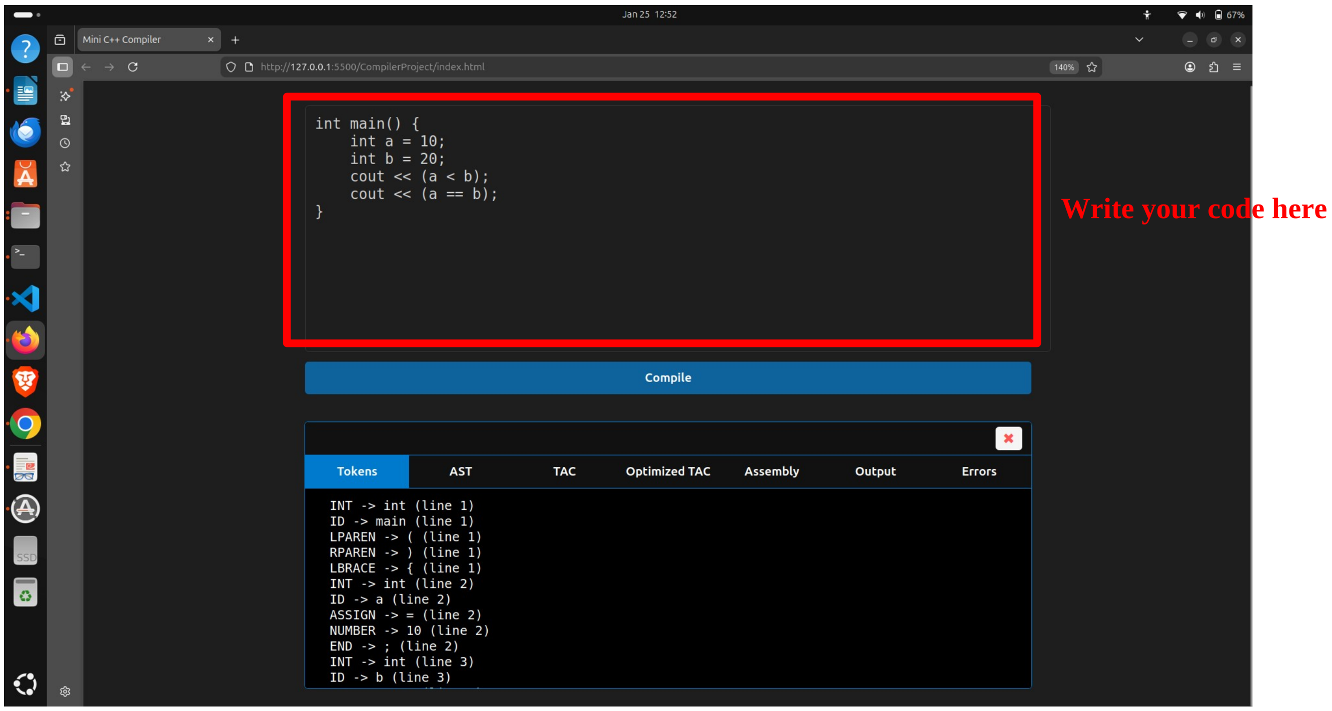
- Tokenized output

- Three Address Code (TAC)

- Assembly code

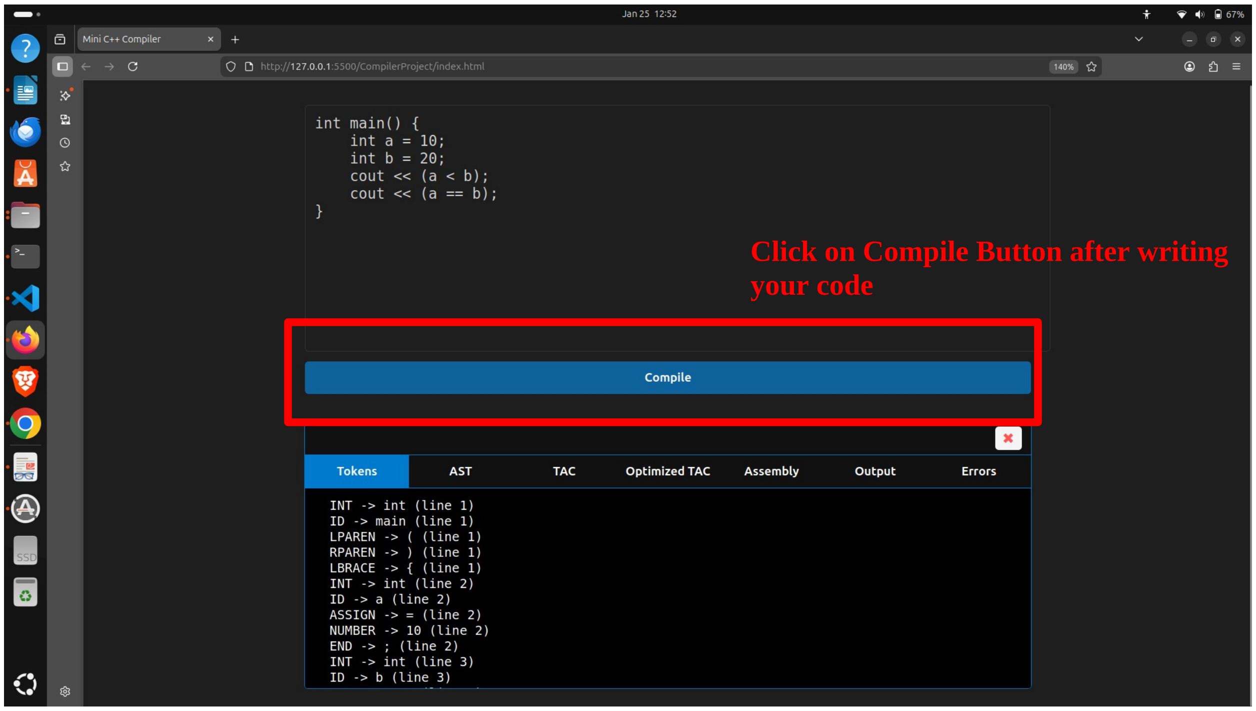
- Program output

- Any error messages (if applicable)



```
int main() {
    int a = 10;
    int b = 20;
    cout << (a < b);
    cout << (a == b);
}
```

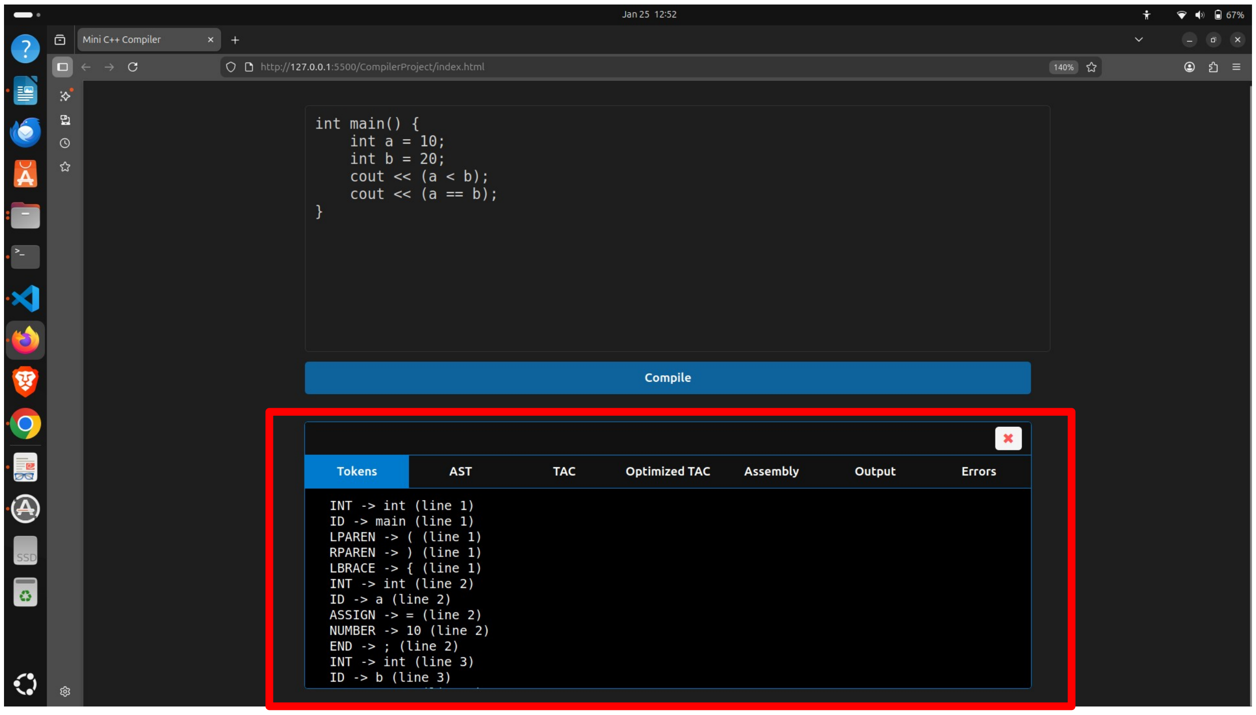
```
INT -> int (line 1)
ID -> main (line 1)
LPAREN -> ( (line 1)
RPAREN -> ) (line 1)
LBRACE -> { (line 1)
INT -> int (line 2)
ID -> a (line 2)
ASSIGN -> = (line 2)
NUMBER -> 10 (line 2)
END -> ; (line 2)
INT -> int (line 3)
ID -> b (line 3)
```



```
int main() {
    int a = 10;
    int b = 20;
    cout << (a < b);
    cout << (a == b);
}
```

Click on Compile Button after writing your code

```
INT -> int (line 1)
ID -> main (line 1)
LPAREN -> ( (line 1)
RPAREN -> ) (line 1)
LBRACE -> { (line 1)
INT -> int (line 2)
ID -> a (line 2)
ASSIGN -> = (line 2)
NUMBER -> 10 (line 2)
END -> ; (line 2)
INT -> int (line 3)
ID -> b (line 3)
```



```
int main() {
    int a = 10;
    int b = 20;
    cout << (a < b);
    cout << (a == b);
}
```

```
INT -> int (line 1)
ID -> main (line 1)
LPAREN -> ( (line 1)
RPAREN -> ) (line 1)
LBRACE -> { (line 1)
INT -> int (line 2)
ID -> a (line 2)
ASSIGN -> = (line 2)
NUMBER -> 10 (line 2)
END -> ; (line 2)
INT -> int (line 3)
ID -> b (line 3)
```

The below output will show you Tokens, AST, TAC, Optimized TAC, Assembly Code, Output. Error if present.

Scope of Project :

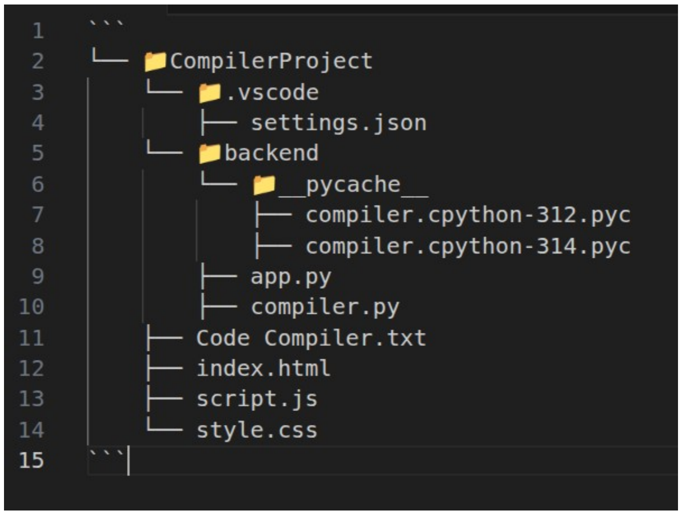
Currently Supported :

- Complete program structure with main() function
- Data types: int, float, char, string, auto (with type inference)
- Variable declarations and assignments with type checking
- Arithmetic operators: +, -, *, /, %
- Relational operators: ==, !=, <, >, <=, >=
- Control flow: if-else statements, while loops
- Output via cout<< with multiple chained expressions
- Type inference for auto-declared variables
- Constant folding optimization
- Dead code elimination optimization
- Comprehensive error reporting across all phases

4. System Design :

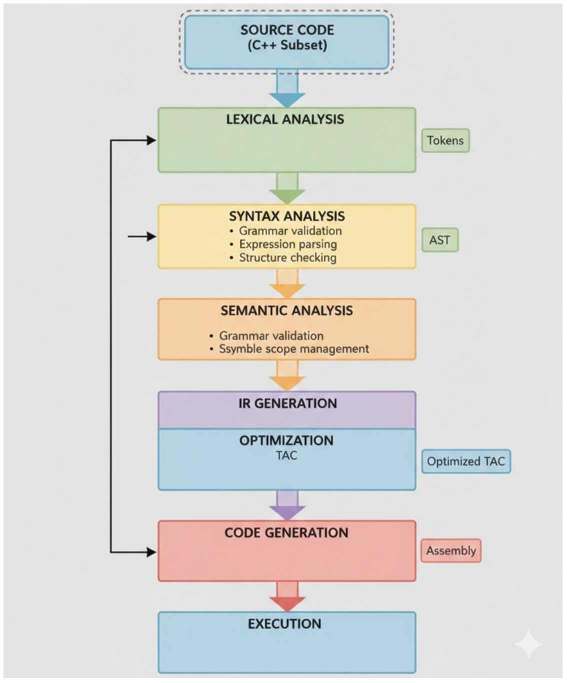
4.1 Overall Architecture:

The compiler follows a modular design with clear separation between front-end (analysis) and back-end (synthesis) phases. The front-end processes the source code through lexical, syntax, and semantic analysis to produce a verified intermediate representation. The back-end then optimizes this representation and generates target code.



```
```
└── 📁CompilerProject
    └── 📁.vscode
        ├── settings.json
    └── 📁backend
        └── 📁__pycache__
            ├── compiler.cpython-312.pyc
            ├── compiler.cpython-314.pyc
        ├── app.py
        ├── compiler.py
    ├── Code Compiler.txt
    ├── index.html
    ├── script.js
    └── style.css
```
```

4.2 Data Flow Diagram:



5. Modules Description

5.1 Lexical Analysis

Purpose : Convert source code to tokens.

Implementation Details :

- Uses Python's `**re**` module with a master regular expression pattern
- Tokenizes keywords, identifiers, literals, operators, and delimiters
- Handles line numbers for accurate error reporting
- Processes string literals with escape sequence support

Token Categories :

Keywords : int, float, char, string, auto, if, else, while, etc.

Identifiers : Variable names starting with letter/underscore

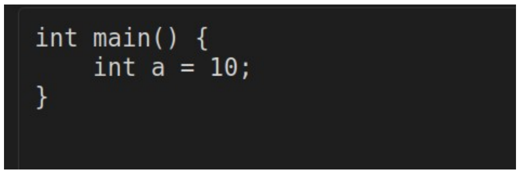
Literals : Numbers (int/float), strings, characters

Operators : Arithmetic (+, -, *, /, %), relational (==, !=, <, >, etc.)

Delimiters : Parentheses, braces, semicolons, commas

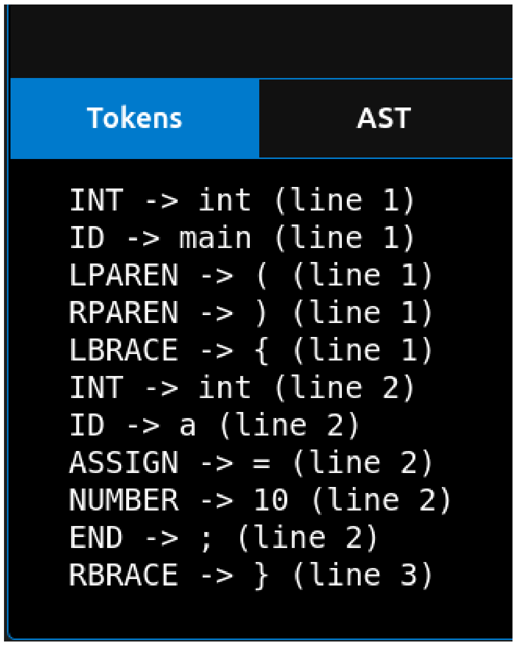
Special : Shift operator (<<) for cout statements

Implementation Input :



```
int main() {
    int a = 10;
}
```

Implementation Output :



```
INT -> int (line 1)
ID -> main (line 1)
LPAREN -> ( (line 1)
RPAREN -> ) (line 1)
LBRACE -> { (line 1)
INT -> int (line 2)
ID -> a (line 2)
ASSIGN -> = (line 2)
NUMBER -> 10 (line 2)
END -> ; (line 2)
RBRACE -> } (line 3)
```

5.2 Syntax Analyzer :

Purpose : Validate grammatical structure and build Abstract Syntax Tree (AST).

Implementation Details :

- Recursive-descent LL(1) parsing strategy
- Each non-terminal has a corresponding parsing function
- Maintains position in token stream with `**pos**` counter
- Uses `**peek()**` and `**advance()**` for token lookahead

Grammar Rules :

Program structure : Sequence of statements within main() function

Variable declarations : `type identifier [= expression];`

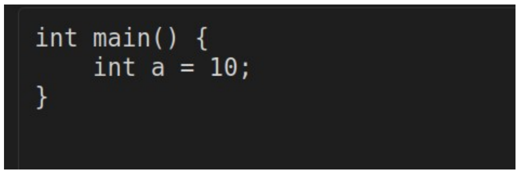
Assignment statements : `identifier = expression;`

Control flow : if-else statements, while loops

Output statements : `cout << expression << ...;`

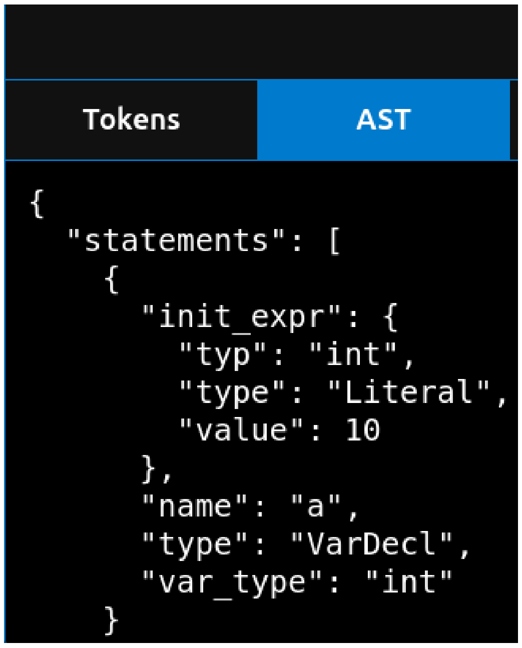
Expressions : Handled with precedence climbing

Example Input :



```
int main() {
    int a = 10;
}
```

Example Output :



```
{
  "statements": [
    {
      "init_expr": {
        "typ": "int",
        "type": "Literal",
        "value": 10
      },
      "name": "a",
      "type": "VarDecl",
      "var_type": "int"
    }
```

5.3 Semantic Analyzer :

Purpose : Validate meaning and types.

Implementation Details :

- Maintains symbol table mapping variable names to types
- Performs type inference for variables declared with **`auto`**
- Checks for undeclared variables and duplicate declarations
- Validates type compatibility for assignments and operations

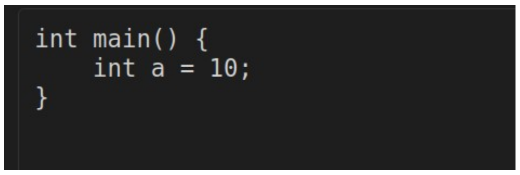
Symbol Table Structure :

```
symbol_table = {
   'variable_name': 'data_type',
   'x': 'int',
   'pi': 'float',
   'name': 'string'
}
```

Type Inference Capabilities :

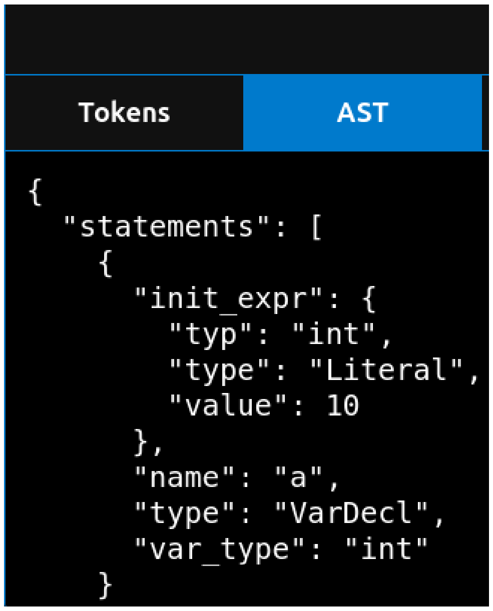
- Infers types from initializer expressions
- Handles type promotion (int → float)
- Validates conditional expressions for if/while statements
- Ensures consistent type usage throughout the program

Example Input :



```
int main() {
    int a = 10;
}
```

Example Output :



```
{
  "statements": [
    {
      "init_expr": {
        "typ": "int",
        "type": "Literal",
        "value": 10
      },
      "name": "a",
      "type": "VarDecl",
      "var_type": "int"
    }
```

5.4 Intermediate Code Generation (TAC)

Purpose : Generate Three Address Code.

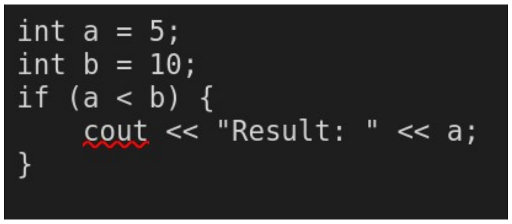
Implementation Details :

- Transforms AST into TAC instructions
- Creates temporary variables for intermediate results
- Handles control flow with labels and jumps
- Generates instructions for arithmetic, relational, and I/O operations

TAC Instruction Types :

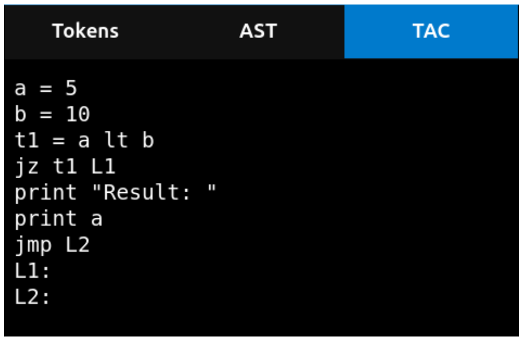
- **Assignment :** `x = y`
- **Arithmetic :** `t1 = a add b`
- **Relational :** `t2 = x lt y`
- **Labels :** `L1:`
- **Jumps :** `jmp L1`, `jz condition L1`
- **Print :** `print value`

Example :



```
int a = 5;
int b = 10;
if (a < b) {
    cout << "Result: " << a;
}
```

Generated TAC :



```
a = 5
b = 10
t1 = a lt b
jz t1 L1
print "Result: "
print a
jmp L2
L1:
L2:
```

5.5 Optimization Phase :

Purpose : Improve code efficiency through transformations.

Implemented Optimizations :

Constant Folding :

- Evaluates constant expressions at compile time
- Replaces operations with constant results with direct assignments
- Handles arithmetic and relational operations

Example :

Original : t1 = 5 + 10

Optimized : t1 = 15

Dead Code Elimination :

- Removes assignments to variables that are never used
- Uses backward analysis to identify live variables
- Preserves side-effecting operations (print, jumps)

Example :

Original :

t1 = a add b

t2 = c mul d

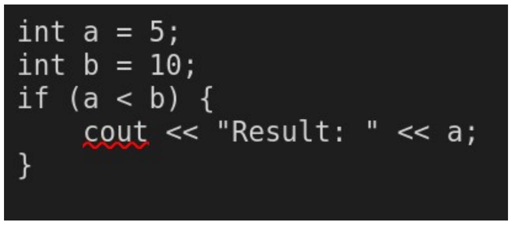
x = t1

Optimized :

t1 = a add b

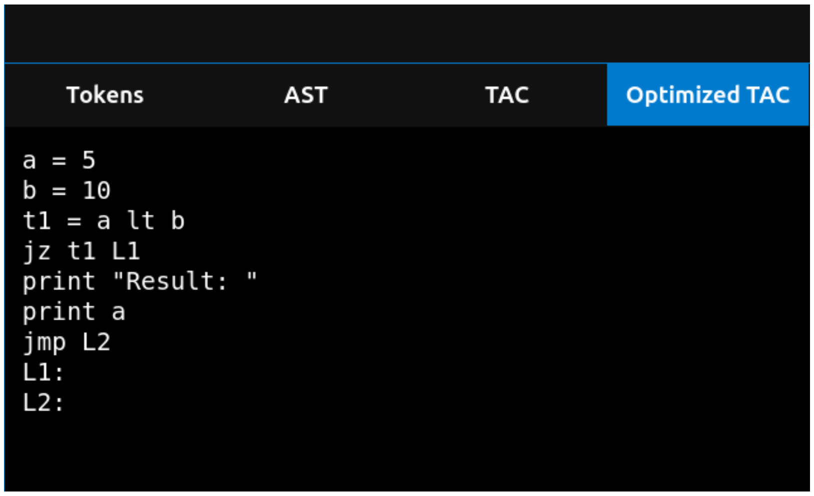
x = t1

Example Input :



```
int a = 5;
int b = 10;
if (a < b) {
    cout << "Result: " << a;
}
```

Example Output :



```
a = 5
b = 10
t1 = a lt b
jz t1 L1
print "Result: "
print a
jmp L2
L1:
L2:
```

5.6 Assembly Generator :

Purpose : Convert optimized TAC to assembly-like instructions.

Implementation Details :

- Translates TAC instructions to register-based assembly
- Uses single general-purpose register (R1) model

- Handles arithmetic operations with load-compute-store pattern
- Generates labels and jumps for control flow

Instruction Mapping :

Assignment : `x = y` → `MOV x, y`

Arithmetic : `t1 = a add b` →

MOV R1, a

ADD R1, b

MOV t1, R1

Labels : `L1:` → `L1:`

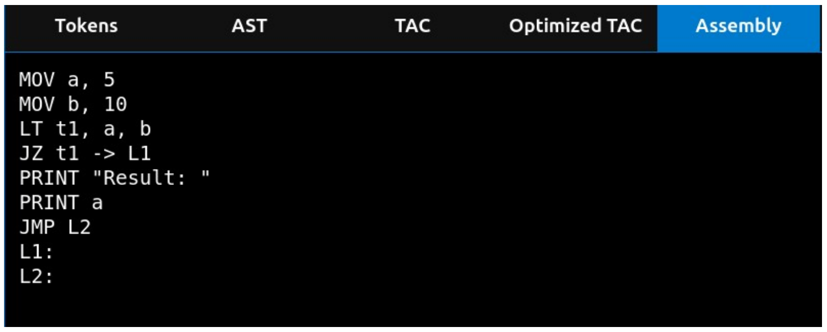
Jumps : `jmp L1` → `JMP L1`

Print : `print x` → `PRINT x`

Example Input :

```
int a = 5;
int b = 10;
if (a < b) {
    cout << "Result: " << a;
}
```

Example Output :



```
MOV a, 5
MOV b, 10
LT t1, a, b
JZ t1 -> L1
PRINT "Result: "
PRINT a
JMP L2
L1:
L2:
```

6. C++ language rules implemented :

6.1 Program Structure

```
int main() {
    // statements
}
```

6.2 Variable Declaration Rules

```
int x;           // √ Valid
float y = 3.14;  // √ Valid
auto z = 5 + 3;  // √ Valid (type inferred as int)
string name = "John"; // √ Valid
int 123var;      // ✗ Invalid (starts with number)
```

6.3 Type Conversion Rules

1. Implicit Conversion :

int → float allowed

float f = 10; // √ 10 converted to 10.0

2. No Conversion :

string ↔ int not allowed

string s = 10; // ✗ Type mismatch error

6.4 Expression Evaluation Rules

Operator Precedence : */% > +-> == != < > <= >=

Left Associative : All operators

Parentheses : Override precedence

7. Tasks Perform,ed by the Compiler :

7.1 Compilation Tasks

Source Code Reading : Read C++ subset code

Tokenization : Break into lexical units

Syntax Validation : Check grammar rules

Semantic Checking : Verify types and declarations

Intermediate Code Generation : Create TAC

Code Optimization : Constant folding, dead code elimination

Target Code Generation : Produce assembly-like output

Error Reporting : Identify and report issues

8. Expressions Compiled :

8.1 Arithmetic Expressions

```
int result = a + b * c;      // √
float area = 3.14 * r * r;  // √
int mod = value % divisor;   // √
```

8.2 Relational Expressions

```
bool check = (x > y);        // √
if (score >= 50) { ... }     // √
```

8.3 Control Flow Statements

```
if (a < b) { ... } else { ... }  // √
while (condition) { ... }        // √
```

8.4 Output Expressions

```
cout << "Value: " << x << " Result: " << y; // √
```

9. Conditional Statements :

9.1 Currently Supported

```
if (condition) {
   // statements
} else {
   // statements
}
```

While Condition :

```
while (condition) {
   // statements
}
```

9.2 Implementation Details

- If statements generate two labels (else and end)
- While loops generate start and end labels
- Condition evaluation followed by conditional jumps
- Proper scoping of variables within blocks

10. Operations Implemented :

10.1 Arithmetic Operators

Operator	Name	Example	CPU
+	Addition	a+b	1 cycle
-	Subtraction	a-b	1 cycle
*	Multiplication	a*b	3-10 cycles
/	Division	a/b	10-40 cycles
%	Modulus	a%b	10-40 cycles

10.2 Relational Operators

Operator	Name	Example
==	Equal to	a == b
!==	Not equal	a != b
<	Less than	a < b
>	Greater than	a > b
<=	Less than or above	a <= b

10.3 I/O Operator

Operator	Name	Example
<<	Insertion	cout << value

11. Error Handling Capabilities :

11.1 Lexical Errors

int 123var; // Error: invalid identifier

float price@; // Error: invalid character

Error Message :`Lexical error (line X): invalid identifier '123var'`

11.2 Syntax Errors

int x = 10   // Error: missing semicolon

float y = (10+5 // Error: mismatched parentheses

Error Message :`Syntax error (line X): missing semicolon`

11.3 Semantic Errors

int x = "hello";   // Error: type mismatch

result = a + b;   // Error: undeclared variables

int x; int x;   // Error: redeclaration

Error Message : `Semantic error (line X): type mismatch: cannot assign string to int`

12. Input & Output Examples :

12.1 Sample Input Program

```
int main() {
   int a = 5;
   int b = 10;
   int c = a + b * 2;
   int d = c * 2;
   if (a > 0) {
      cout << "Result: " << d;
   }}
```

12.2 Compiler Output

TOKENS :

```
INT -> int (line 1)
ID -> main (line 1)
LPAREN -> ( (line 1)
RPAREN -> ) (line 1)
LBRACE -> { (line 1)
```

```
INT -> int (line 2)
ID -> a (line 2)
ASSIGN -> = (line 2)
NUMBER -> 5 (line 2)
END -> ; (line 2)
```

THREE ADDRESS CODE (unoptimized) :

```
a = 5
b = 10
t1 = b * 2
t2 = a + t1
c = t2
t3 = c * 2
d = t3
t4 = a > 0
jz t4 L2
print "Result: "
print d
jmp L3
L2:
L3:
```

THREE ADDRESS CODE (optimized) :

```
a = 5
b = 10
t1 = 20
t2 = a + t1
c = t2
t3 = c * 2
d = t3
t4 = a > 0
jz t4 L2
```

```
print "Result: "
print d
jmp L3
L2:
L3:
```

ASSEMBLY :

```
MOV a, 5
MOV b, 10
MOV t1, 20
MOV R1, a
ADD R1, t1
MOV t2, R1
MOV R1, c
MOV t3, R1
MOV R1, a
GT R1, 0
MOV t4, R1
JZ t4 -> L2
PRINT "Result: "
PRINT d
JMP L3
L2:
L3:
```

OUTPUT :

```
Result : 50
```

Conclusion :

The Mini Optimizing Compiler (MOC) successfully demonstrates the core phases of compilation for a practical subset of C++ with a strong emphasis on code optimization. Implemented in Python, the compiler performs lexical analysis, syntax and semantic checking, generates Three Address Code (TAC), applies optimization techniques, and produces assembly-like output with clear, line-based error reporting.

The compiler's focus on optimization techniques, particularly constant folding and dead code elimination, demonstrates how compile time analysis can improve program efficiency without altering program behavior. The modular design with clear separation between front-end and back-end phases creates a maintainable and extensible foundation for future enhancements.

The comprehensive error reporting system and support for multiple programming constructs make this compiler a valuable educational tool for understanding both the theoretical foundations and practical implementation details of modern compiler technology.