A Mini Compiler for the C++ programming language, focuses on generating optimized intermediate code and target code for the language for specific constructs.
It works for constructs such as conditional statements and loops
The main functionality of the project is to generate an optimized intermediate code and target code for the given C++ source code.

This is done using the following steps:
i) Generate symbol table after performing expression evaluation
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code followed by corresponding quadruples
iv) Perform Code Optimization
v) Generate Target Code.

The main tools used in the project include **LEX** which identifies pre-defined patterns and generates tokens for the patterns matched and **YACC** which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code for the source code. **Python** is used to optimize the intermediate code generated by the parser and generate Assembly Code.

ARCHITECTURE OF LANGUAGE

C++ constructs implemented:
1. if
2. while loop

- Arithmetic expressions with +, -, *, /, ++, -- are handled
- Boolean expressions with >,<,>=,<=,== are handled
- Error handling reports undeclared variables
- Error handling also reports syntax errors with line numbers

LITERATURE SURVEY AND OTHER REFERENCES

https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

CONTEXT FREE GRAMMAR

```
%token IDENTIFIER STRING_LITERAL
%token INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN

%token CHAR INT VOID
%token IF WHILE
%Start s

s
      : main_dec
      | s main_dec
      ;

main_dec                              ## int main(){ .. }
      : INT MAIN '(' ')' compound_statement
      | VOID MAIN '(' ')' compound_statement
      | declaration
      | headers
      ;

headers                                    ## #include <myfile.h>
      : HASH INCLUDE HEADER_LITERAL                ## #include
<IOSTREAM>
      | HASH INCLUDE '<' libraries '>'
      ;

libraries
      : IOSTREAM
      | STDLIB
      | MATH
      | STRING
      | TIME
      ;

compound_statement
      : '{' '}'
```

```
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      ;

declaration_list
      : declaration
      | declaration_list declaration
      ;

declaration
      : type_specifier init_declarator_list ';'        #int
a,b,c,d=10;
      ;

init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator    ## a=10 , b=20,
....


init_declarator
      : IDENTIFIER '=' assignment_expression
      | IDENTIFIER


statement_list
      : statement
      | statement_list statement


statement
      : compound_statement
      | expression_statement
      | iteration_statement
      | selection_statement
      ;

type_specifier
      : VOID
      | CHAR
      | INT
      ;
      ;
```

```
assignment_expression
	: equality_expression
	| unary_expression assignment_operator assignment_expression
	;

assignment_operator
	: '='
	| ADD_ASSIGN           ## +=
	| SUB_ASSIGN           ## -=
	;

expression_statement
	: ';'
	| expression ';'
	;

expression
	: assignment_expression
	| expression ',' assignment_expression
	;

iteration_statement
	: WHILE '(' expression ')' statement
	;
selection_statement
	: IF '(' expression ')' statement


equality_expression
	: relational_expression
	| equality_expression EQ_OP relational_expression
	| equality_expression NE_OP relational_expression
	;

multiplicative_expression                            ##  E-> T|E+T
	: unary_expression                             ##  T-> F|T*F

	| multiplicative_expression '*' unary_expression  ##  F-> num
	| multiplicative_expression '/' unary_expression
	;

additive_expression
	: multiplicative_expression
```

```
      | additive_expression '+' multiplicative_expression
      | additive_expression '-' multiplicative_expression
      ;

relational_expression
      : additive_expression
      | relational_expression '<' additive_expression
      | relational_expression '>' additive_expression
      | relational_expression LE_OP additive_expression
      | relational_expression GE_OP additive_expression
      ;

unary_expression
      : postfix_expression
      | unary_operator unary_expression    ## !a ++a --a etc
      ;

postfix_expression
      : primary_expression             ##a++ b-- etc
      | postfix_expression '(' ')'
      | postfix_expression '.' IDENTIFIER
      | postfix_expression INC_OP
      | postfix_expression DEC_OP
      ;

primary_expression
      : IDENTIFIER
      | INTEGER_LITERAL
      | STRING_LITERAL
      | FLOAT_LITERAL
      | CHARACTER_LITERAL
      | '(' expression ')'
      ;

unary_operator
      : '+'
      | '-'
      | '&'
      | '|'
      | '!'
      ;
```

DESIGN STRATEGY

- SYMBOL TABLE CREATION

  It is a data structure that stores information about the occurrence of identifiers. Every new variable with a different scope encountered into the program is entered into the symbol table. Else, the current line is added to the current variable's and scope's entry.

- ABSTRACT SYNTAX TREE

  It is a tree representation of the abstract syntactic structure of source code. A binary tree is implemented to represent Abstract Syntax Tree. Each subtree's root is passed onto the parent with the help of $$. The tree is printed in pre-order format

- INTERMEDIATE CODE GENERATION

  The intermediate code is generated along with the parsing of the grammar. It is stored in Quadruple format which is then used for code optimization

- CODE OPTIMIZATION

  To improve efficiency, code optimization is done on intermediate code. We have implemented elimination of common subexpressions, constant folding and dead code elimination.

- ERROR HANDLING

  In case of syntax error, the error is displayed along with the line number.

- TARGET CODE GENERATION

  The optimised ICG is passed as input to a python script resulting in assembly code.

IMPLEMENTATION DETAILS

Symbol Table Creation

- A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurrence, the category of the token (constant, variable, keyword, operator),the value that it holds the datatype.
- As each line is parsed, the actions associated with the grammar rules is executed.
- Expressions are evaluated and the values of the used variables are updated accordingly.
- At the end of the parsing, the updated symbol table is displayed.

Abstract Syntax Tree

A tree structure representing the syntactical flow of the code is generated in this phase. For expressions, associativity is indicated using the %left and %right fields. Precedence of operations - last rule

```
struct node{
	char token[20];
	char name[20];
	int dtype;
	int scope;
	int lineno;
	int valid;
	union value{
		float f;
		int i;
		char c;
	}val;

	struct node *link;

}*first = NULL, *tmp, *crt, *lhs;

typedef struct Node{
	struct Node *left;
	struct Node *right;
```

```
	char token[100];
	struct Node *val;
	int level;
}Node;

typedef struct tree_stack{
	Node *node;
	struct tree_stack *next;
}tree_stack;
```

Intermediate Code Generation (ICG)

Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.
Three-Address Code –
A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have an address (memory location).

Example – The three address code for the expression a + b * c + d :

```
T1 = b * c
T2 = a + T 1
T3 = T2 + d
```

T1, T2, T3 are temporary variables.
The data structure used to represent, Three address Code, is Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

Code Optimization

The Machine Independent Optimization techniques used are:

- Eliminating Common Subexpression
- Constant Folding
- Dead Code Elimination

Target Code Generation

Target/Assembly Code is generated from the optimized Intermediate Code Generation phase's output.

Commands to execute the code:

AST:

```
lex ast.l
yacc -d ast.y
gcc lex.yy.c y.tab.c -ll -ly -o ast.o
./ast.o < input.cpp
```

ICG:

```
lex icg.l
yacc -d icg.y
gcc lex.yy.c y.tab.c -ll -ly -o icg.o
./icg.o < input.cpp
```

Code Optimization:

```
python optimize.py
```

Target Code Generation:

```
python generate_target.py icg_while.txt
```

SNAPSHOTS

Symbol Table Creation:

```c
#include<stdio.h>
int main(){

    int x, y;
    x=10;
    y = x*2 + 3; //Value of 'y' here is 23
    if(y>0){
        int z;
        z = x + y;
        /* Here, z has local scope.
        But it can access 'x' and 'y' from outside
        and its value will be 33*/
        z = x + y;
    }
}
```

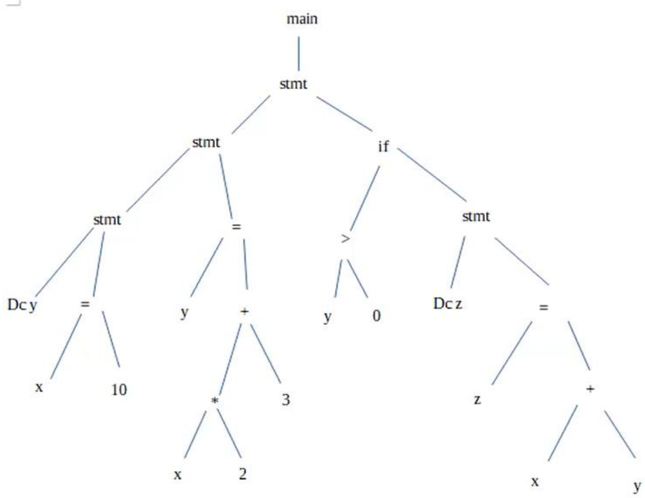
Symbol Table

Symbol	Name	Type	Scope	Line Number	Value
identifier	x	int	1	4	10
identifier	y	int	1	4	23
identifier	z	int	2	8	33

Abstract Syntax Tree


```c
#include<stdio.h>
int main(){

    int x, y;
    x=10;
    y = x*2 + 3; //Value of 'y' here is 23
    if(y>0){
        int z;
        z = x + y;
        /* Here, z has local scope.
        But it can access 'x' and 'y' from outside
        and its value will be 33*/
        z = x + y;
    }
}
```

```
Abstract Syntax Tree
                                              main

                                         stmt

                                    stmt    if

                               stmt    =    >    stmt

                          Dc y    =    y    +    y    0    Dc z    =

                     x    10    *    3    z    +

                x    2    x    y
```




```
Preorder Traversal
main    ( stmt    ( stmt    ( stmt   Dc y    ( =   x   10   )   )    ( =   y    ( +    ( *   x   2   )   3   )   )   )    ( if    ( >   y   0   )
 ( stmt   Dc z    ( =   z    ( +   x   y   )   )   )   )   )
```

Intermediate Code Generation:

```cpp
#include<iostream>
int main()
{
    int i=0;
    int a=11;
    while(i<3){
        a++;
        i++;
    }
    return 0;
}
```

```
Quadruple Format

        Op              opr1            opr2            Result
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        =               0                               i
        =               11                              a
        Label                                           L0
        <               i               3               t0
        ifFalse         t0                              L1
        +               a               1               t1
        =               t1                              a
        +               i               1               t2
        =               t2                              i
        goto                                            L0
        Label                                           L1


Intermediate Code

i = 0
a = 11
L0:
t0 = i < 3
ifFalse t0 goto L1
t1 = a + 1
a = t1
t2 = i + 1
i = t2
goto L0
L1:
```

Code Optimization:

```
t0 = 3 + 1
a = t0
c = a + 3
d = a + 3
t1 = t0
t2 = t1
```

```
manoj@Manoj: /mnt/d/Projects/Compiler Design/Code Optimization
        a = t0
        c = a + 3
        d = a + 3
        t1 = t0
        t2 = t1


ICG after eliminating common subexpressions:

        t0 = 3 + 1
        a = t0
        c = a + 3
        d = c
        t1 = t0
        t2 = t1


ICG after constant folding:

        t0 = 4
        a = t0
        c = a + 3
        d = c
        t1 = t0
        t2 = t1


Optimized ICG after dead code elimination:

        t0 = 4
        a = t0
        c = a + 3
        d = c


Optimization done by eliminating 2 lines.


manoj@Manoj:/mnt/d/Projects/Compiler Design/Code Optimization$
```

Target Code Generation:

```
i = 0
a = 11
j = 0
t0 = a + i
t1 = t0 * a
j = t1
```

```
MOV R0 0
STR R0 i
MOV R1 11
STR R1 a
MOV R2 0
STR R2 j
ADD R3 R1 R0
MUL R4 R3 R1
STR R4 j
```

```
i = 0
a = 11
L0:
t0 = i < 3
ifFalse t0 goto L1
t1 = a + 1
a = t1
t2 = i + 1
i = t2
goto L0
L1:
```

```
        MOV R0 0
        STR R0 i
        MOV R1 11
        STR R1 a
L0:
        CMP R0 3
        BGE L1
        ADD R2 R1 1
        STR R2 a
        ADD R3 R0 1
        STR R3 i
        B L0
L1:
```

RESULTS AND POSSIBLE SHORTCOMINGS:

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an optimized intermediate code and target code, given a C++ code as input.
There are a few shortcomings with respect to our implementation.
The symbol table structure is same across all types of tokens (constants, identifiers and operators). This leads to some fields being empty for some of the tokens. This can be optimized by using a better representation.
The Code optimizer does not work well when propagating constants across branches (At if statements and loops). It works well only in sequential programs. This needs to be rectified.

FUTURE ENHANCEMENTS:

As mentioned above, we can use separate structures for the different types of tokens and then declare a union of these structures. This way, memory will be properly utilized.

For constant propagation at branches, we need to implement SSA form of the code. This will work well in all cases and yield the right output.