Report on

"Mini Compiler on C++"

*Submitted in partial fulfillment of the requirements for **Sem VI***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Taran Singhania	**PES1201800333**
Harsha Kulkarni	**PES1201802000**
Rahul Kata	**PES1201802018**

Under the guidance of

Mahesh H.B.
Assistant Professor
PES University, Bengaluru

January – May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

Introduction

This compiler was built for a subset of the C++ language. The constructs assigned to us were: if, if-else and for.
The compiler takes in a C++ program as input and outputs the optimized code, symbol table, etc.

Multiple implementation stages were involved to implement our final result. These include syntax analysis, semantic analysis, Intermediate Code Generation, and finally, Code Optimization.

For the most part, Lex and Yacc in C were used to implement the compiler, along with python in the very last stage - to convert Intermediate Three Address Code to Optimized code.

Architecture of the language

Syntax:

The following constructs have been implemented:

- If
- If - else
- For loop

This includes any arbitrary depth of nesting and any arbitrary number of permutations for each of these constructs and the way they fit into C++ language constructs like code blocks, operations, etc.

Arithmetic Operators handled:

- +
- -
- *
- /
- ++ (pre and post)
- -- (pre and post)
- %

Relational Operators handled:

- >
- >=
- <
- <=
- ==
- !=
- &&
- ||
- !

Shorthand Operators handled:

- +=
- -=
- *=
- /=
- %=
- <<=
- >>=

Semantics:
Error handling:
The following are reported as errors:

- Undeclared variables
- Declared variables that are out of scope
- Declared variables that have gone out of scope and have been killed and hence no longer valid
- Mismatch of data types
- Redeclaration of variables
- Wrong syntax for any of the given constructs
- Wrong syntax for any of the core C++ concepts
- Wrong nesting of parentheses
- Wrong usage of parentheses with specific constructs
- Mismatched parentheses
- Wrong syntax in nested arithmetic operations, in assignments.

Each error is reported with a verbose explanation of what the mistake is, along with the line number at which it occurred. Due to the error handling mechanism that is built into the compiler, the compiler continues to parse through the rest of the file.

LITERATURE SURVEY

A large part of the project was implemented by using the material provided to us in the CD-2021 folder and the demonstrations shown in class, which were largely sufficient to what we sought to build. Material on the internet that explained the usage of lex and yacc, documentation, etc. were used additionally.

CONTEXT FREE GRAMMAR

```
S: START

START: T_INCLUDE Prog
| Prog
;

Prog: DECL BODY
| BODY
;

BODY: T_MAIN T_OFB STMT T_CFB
| T_VMAIN T_OFB STMT T_CFB
| T_CLASS T_ID T_OFB CLASS_STMT T_CFB BODY
| FUNC BODY
|
;

CLASS_STMT: T_PUBLIC STMT CLASS_STMT
|T_PRIVATE STMT CLASS_STMT
|T_PROTECTED STMT CLASS_STMT
|
;
```

```
STMT: DECL STMT
|ASSIGN_EXPR STMT
|T_IN INOUT T_Terminator STMT
|T_OUT INOUT T_Terminator STMT
|FORSTMT STMT
|SELECTSTMT STMT
|JUMPSTMT STMT
|T_OFB STMT T_CFB STMT
|UNARYEXP T_Terminator STMT
|FUNC STMT
|FUNCALL STMT
|ARRAYASSGN STMT
|
;

ARRAYASSGN: T_ID T_OSB VALUE T_CSB T_EQ VALUE T_Terminator
;

VALUE: T_ID
|T_NUM
;


FUNC: TYPE T_ID T_OB PARAMLIST T_CB T_OFB STMT T_CFB {}
;

PARAMLIST: TYPE T_ID PARAMLIST
| TYPE T_ID T_EQ E PARAMLIST
| T_COMMA T_ID T_EQ E PARAMLIST
| T_COMMA TYPE T_ID PARAMLIST
|
;

FUNCALL: T_ID T_OB F_PARAMLIST T_CB T_Terminator
;
```

```
F_PARAMLIST: T_COMMA T_ID F_PARAMLIST
|T_ID F_PARAMLIST
|T_COMMA T_ID T_EQ E F_PARAMLIST
|
;

INOUT: T_STRIN T_ID
|T_STROUT T_ID INOUT
|T_STROUT T_ENDL INOUT
|T_STROUT T_STRING INOUT
|
;

TYPE: T_INT
|T_FLOAT
|T_CHAR
|T_DOUBLE
|T_VOID
|T_STR
;

DECL: TYPE T_ID T_EQ E VARLIST  T_Terminator
| TYPE T_ID VARLIST T_Terminator
| TYPE T_ARRAY T_Terminator
| TYPE T_ARRAY T_EQ T_OFB ARRELE T_CFB T_Terminator
;

ARRELE: T_NUM T_COMMA ARRELE
| T_STRING T_COMMA ARRELE
| T_NUM
| T_STRING
|
;
```

```
VARLIST: T_COMMA T_ID VARLIST
|T_ID
|T_COMMA T_ID T_EQ E VARLIST
|
;

ASSIGN_EXPR: T_ID ASSIGNOP E T_Terminator
;

E: E T_ADD T
|E T_SUB T
|T
;

T: T T_MUL F
|T T_DIV F
|F

F: T_ID
|T_NUM
| T_OB E T_CB
| BOOL
| UNARYEXP
|T_STRING
;

FORSTMT: T_FOR T_OB INITFOR COND T_Terminator ITRCHANGE T_CB T_OFB STMT
T_CFB
;

INITFOR: T_ID T_Terminator
|ASSIGN_EXPR
|DECL
|T_Terminator
;
```

```
COND: OPERATION
;

OPERATION: E LOGICOP E
|E RELOP E
|E BITOP E
|E RELOP E LOGICOP E RELOP E
|T_NOT T_OB E RELOP E T_CB
|T_NOT E RELOP E
|T_NOT E
|E
|
;

ITRCHANGE: UNARYEXP
| ASSIGN_EXPR
|
;

SELECTSTMT: T_IF T_OB COND T_CB T_OFB STMT T_CFB ELSEBODY
;

ELSEBODY: T_ELSE T_OFB STMT T_CFB
|
;

JUMPSTMT: T_RETURN T_ID T_Terminator
| T_RETURN T_NUM T_Terminator
| T_RETURN T_STRING T_Terminator
| T_RETURN T_Terminator
| T_CONTINUE T_Terminator
| T_BREAK T_Terminator
;


UNARYEXP: T_INC T_ID
```

```
| T_ID T_INC
| T_DEC T_ID
| T_ID T_DEC
;

LOGICOP: T_OR
| T_AND
;

BITOP: T_BAND
| T_BOR
| T_BXOR
;

RELOP: T_DEQ
| T_LT
| T_GT
| T_GTE
| T_LTE
| T_NE
;

ASSIGNOP: T_EQ
| T_ADDEQ
| T_MULEQ
| T_DIVEQ
| T_MODEQ
| T_LSEQ
| T_RSEQ
| T_SUBEQ
;

BOOL: T_TRUE
| T_FALSE
;
```

DESIGN STRATEGY

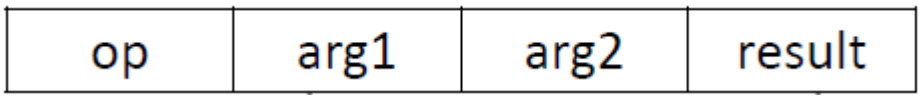
SYMBOL TABLE CREATION :

Symbol table is supposed to hold the relevant information for every identifier, like the identifier name, value, scope, etc.
Naturally, an array of a structure with each element being able to hold the required information would suffice.
The idea is to append to the list while parsing declarations of variables, and then updating their value whenever we encounter arithmetic operations involving numbers getting assigned to the tree. The scope can largely be characterized by the nesting depth (of code blocks) with respect to the global segment of the program (along with more complex handling for statements in looping constructs). Owing to this, we can implement this in the tokenizer using simple counting variables.
We also need to remove entries from the symbol table the moment they go out of scope (reach a level that is lower than the one they were initialized in), so that any further uses would have to have to be declared again, even when the current scope level is above the one the variable was previously initialized in.
But for the sake of demonstration in this project, we have refrained from removing entries in the table (since all variables would go out of scope when the program ends, leaving the table empty), instead opting to use a flag variable indicating the variable's validity at the current point. The idea is that once the scope reaches a lower scope level than the one initialized in, the flag is set to zero. Again, this can be done using simple counting variables.

INTERMEDIATE CODE GENERATION:

Intermediate code generation requires a line by line parsing of the input, with the program state and the necessary values being pushed on to a stack.
Whenever we encounter a construct, we would have to pop the right number of elements and adjust the bookkeeping variables accordingly, and use this information to fill into a template specific to each construct (with branch instructions, labels, etc). With the template being filled in recursively (since we do have a stack), we can represent not only an arbitrary depth of nesting for each of these constructs but also an arbitrary sequence of constructs in such a nesting.

We have also used a quadruple table of the following format:
Operator | Operand1 | Operand2 | Result



to keep track of the ICG.

CODE OPTIMIZATION:
With the three address code being generated in the previous phase, it should be rather easy to identify patterns and usage of variables due to the simple and limited vocabulary and permutations of said vocabulary being allowed.
It would be easy to identify variables and propagate them or their values down to other expressions which use these variables or dependents of these variables. This can be done using a lookup table which we can use to keep track of variables that we have encountered so far.

Due to the rather liberal use of temporary variables that have been used to simplify arithmetic operations, we end with a lot of code that is no longer necessary but is still used. We can eliminate this by taking a look at the variables that are no longer used further down the program, since they have no effect. However, this only applies to temporary variables, and not actual variables since it gets harder to define what a useful variable is.

ERROR HANDLING :
A version of Panic mode error recovery is used throughout the project, i.e., across all the implementation stages. Any code that does not adhere to the rules specified is discarded until the next terminating token (;) and the compiler exits its job after that.

Implementation Details

- **Symbol table generation:**
  The symbol table was represented as an array of nodes that had the following structure

```
typedef struct node{
     int scope;
     char value[1000];
     char name[100];
     char dtype[50];
     int line_num;
     int valid;
}node;

typedef struct table{
     node* head;
}table;

extern node symTable[1000];
```

name - name of the identifier in the symbol table.
value - value of the identifier.
scope - the variable's scope in the program.
line_num - the line number at which the variable is present.
valid - represents the lifetime of the variable, becomes 0 once it goes out of scope and is killed.

The scope of a variable is kept track of by using the number of open braces encountered, and the value field is populated with the variable's value by evaluating the RHS of its assignment expression.
The symbol table was populated using both the tokenization and the parsing phase, where the field values such as scope, name, line number, validity and data type were populated in the former

whereas the value of the identifier was filled in during the parsing phase.
For inserting an identifier's information, the symbol table is parsed to check if there's an identifier with the same name and the scope of the identifier in question that is valid (which would mean that it has gone out of scope and variable redeclaration is necessary).

```
!strcmp(symTable[i].name, name) && symTable[i].scope == scope &&
!symTable[i].valid
```

So, while parsing the table if the above condition is true, the idier is not inserted to the symbol table.

- **Literal Table:**
  The literal table is an extension of the symbol table. In this variation the symbol table only stores the variable(token) name, line number and reference number. The reference number is a pointer to the literal table which stores the data type, size, scope and value also. The table structure for both symbol table(right) and literal table(left) are shown below.

```
typedef struct nodeLiteralTable{
    int ref_no;
    int scope;
    int value;
    int size;
    char name[100];
    char dtype[50];
    int line_num;
    int valid;
}nodeLiteralTable;

typedef struct Ltable{
    nodeLiteralTable* head;
}Ltable;

nodeLiteralTable literalTable[1000];
```

```
typedef struct node{
    int scope;
    char name[100];
    char dtype[50];
    int line_num;
    int valid;
    int ref_no;
}node;

typedef struct table{
    node* head;
}table;

node symTable[1000];
```

- **Intermediate Code Generation:**
  The idea followed in the intermediate code generation was mainly populating a global stack, as and when the constructs were encountered in the parsing rules with the variables, operators and generating the snippet by selectively choosing the contents on the top of the stack and decrementing the stack pointer appropriately.

  The code generation was handled differently for different types of constructs. For example, assignment type of expressions require two values to be present, LHS and RHS, and hence the top two elements of the stack are used. In case of a for loop, the code generation was split into four parts, one for each segment of a for loop for label assignment and condition checking, the code of the update logic and the main body was generated before these segments were encountered, owing to the bottom-up nature of the parser.
  For nested conditions of the looping constructs, we mainly kept track of the number of labels required by a particular type, for example a for loop requires 4, and pushed them into the global stack, which was used by the present looping construct to construct the ICG labels and the branch logic, and then pop those same set of labels. This would be repeated for the nested constructs that were present.

  The quadruple table was implemented as a linked list of structures providing capability to expand as much as we want. The structure is as shown below:

```
typedef struct quadruples{
   char op[100];
   char arg1[20];
   char arg2[20];
   char res[20];
   struct quadruples *next;
} quad;
quad *quad_head=NULL;
quad *quad_tail=NULL;
```

To insert into the quadruple table:

```
void insert_quad(char *op,char *arg1,char *arg2,char *res)
{
   quad *new=(quad *)malloc(sizeof(quad));
   strcpy(new->op,op);
   if(arg1!=NULL)
       strcpy(new->arg1,arg1);
   //new->arg2=NULL;
   if(arg2!=NULL)
       strcpy(new->arg2,arg2);
   if(res!=NULL)
       strcpy(new->res,res);
   new->next=NULL;
   if(quad_head==NULL)
   {
       quad_head=new;
       quad_tail=quad_head;
   }
   else
   {
       quad_tail->next=new;
       quad_tail=quad_tail->next;
   }
}
```

- **Code Optimisation:**
  The generated intermediate code in the previous phase was not optimised in the sense that it had a lot of redundant code, there were constant expressions that could be evaluated before compile time and so on. In our case, we pass this IC to a python script which does Constant Folding, Constant Propagation, Common Subexpression Expression, Copy Propagation and Dead Code Elimination.

Constant folding involved evaluating the constant expressions that were present using eval function and modifying the quadruple with operator as assignment, result being the same and first argument as evaluated value.
To implement *Constant Propagation*, each variable with direct assignment is populated in a dictionary as key(variable name) and the value as its value(constant value), when each line is parsed if a key is found as an argument it is replaced by its value.
Copy Propagation is implemented in a similar manner to Constant Propagation but the key and value in the dictionary are both variables. Only for direct variable to variable assignments, a dictionary entry is made. In the subsequent code if the key is found to match with either operand1 or operand2, then it is replaced by its previously assigned variable.
x = y

=	y		x

Dead Code Elimination is done by taking care of many edge cases to cover all the redundant code.If(false) code block will be never executed and on the other hand If(true) code block should be executed mandatorily, hence the select statement is not necessary. All the variables that have been used in select statements(loops) and as parameters in function call should be preserved.

*Common Subexpression*s are eliminated using a helper function replaceAll() which replaces the subsequent common subexpressions with already evaluated variables. Every line is checked with all the subsequent lines to check if it has the same expression(operator and operands) if found, the expressions are replaced with the variable using replaceAll() function and that line is deleted.

RESULTS AND POSSIBLE SHORTCOMINGS

The resulting code is optimised as per requirements and handles the constructs assigned to us. The compiler can handle a variety of core C++ concepts along with for looping constructs and conditional execution statements like if and if-else constructs. The symbol table so generated has the necessary information, handling both the syntactic and semantic discrepancies. The quadruple table displays the ICG in an orderly manner.

The possible shortcomings are the less than robust handling of certain expressions, and inability to recover from errors.
It would be much more beneficial to add more C++ concepts to the subset supported as well.

Another shortcoming would be the implementation of the symbol table which is limited to the size of the array of structs as they are defined before and they are not implemented as linked lists.

SNAPSHOTS

test cpp code for subsequent operations

```
#include <iostream.h>

int main(){
int a = 10;
int b = a + 50;
float z = 10.5 + 1;
z++;
a++;


int y=100;
int c;
c = a + 50;

int e = a + 100 * 100;

foo(a,b,c);

for(int i=0; i<100; i++){
    int d = 12;
}

return 0;
}
```

- symbol table and token generation:

```
SYMBOL TABLE:
Token           Data type       Scope           Value           Line number
a               int             1               10              11
b               int             1               40              12
z               float           1               11.500000               13
y               int             1               100             18
c               int             1               50              19
e               int             1               10100           22
i               int             2               0               26
d               int             3               12              27
```

- literal table

```
SYMBOL TABLE:
Token           Line number             Ref#
a               7                       1
i               15                      2
x               17                      3
a               19                      4
b               20                      5


LITERAL TABLE:
Ref#            Token           Data type       Size    Scope           Value           Line number
1               a               int             4       1               0               7
2               i               int             4       3               0               15
3               x               int             4       3               10              17
4               a               int             4       2               -10             19
5               b               int             4       2               1               20
```

- methods used for ICG of for construct:

```
void for1() //mainly for getting labels ready
{
      label[ltop++] = ++lnum; // label after (for(int i = 0;)) i.e. after
    //    printf("L%d : \n",label[ltop - 1]);
      char lab[26] = "L";
      char temp1[100];
      sprintf(temp1, "%d", label[ltop - 1]);
      strcat(lab, temp1);

      insert_quad("Label",NULL,NULL,lab);
      label[ltop++] = ++lnum; //label for incrementing
      label[ltop++] = ++lnum; //label for body of for loop
      label[ltop++] = ++lnum; //label for statements outside for loop
}
```

```
void for2() //ICG for condition check
{
    strcpy(temp,"t");
    strcat(temp,i_);
    // printf("%s = not %s %s %s\n",temp,st[top-3],st[top-2],st[top-1]);
    char temp1[50] = "not ";
    strcat(temp1, st[top-2]);
    insert_quad(temp1,st[top-3],st[top-1],temp);

    top -= 2;

    // printf("if %s goto L%d\n",temp,label[ltop - 1]);
    char lab2[26] = "L";
    char temp2[100];
    sprintf(temp2, "%d", label[ltop - 1]);
    strcat(lab2, temp2);
    insert_quad("if",temp,NULL,temp2);

    if(i_[1]!='9')
        i_[1]++;
    else
    {
        i_[1] = '0';
        i_[0]++;
    }
    // printf("goto L%d\n", label[ltop - 3]);
    char lab3[26] = "L";
    char temp3[100];
    sprintf(temp3, "%d", label[ltop - 3]);
    strcat(lab3, temp3);
    insert_quad("goto",NULL,NULL,lab3);

//   printf("L%d :\n", label[ltop - 2]);
    char lab4[26] = "L";
  char temp4[100];
  sprintf(temp4, "%d", label[ltop - 2]);
  strcat(lab4, temp4);
  insert_quad("Label",NULL,NULL,lab4);

}
```

```
void for3() //incrementing and executing loop again
{
//   printf("goto L%d\n", label[ltop - 4]);
  char lab[26] = "L";
  char temp1[100];
  sprintf(temp1, "%d", label[ltop - 4]);
  strcat(lab, temp1);
  insert_quad("goto",NULL,NULL,lab);

//   printf("L%d :\n", label[ltop - 3]);
  char lab2[26] = "L";
  char temp2[100];
  sprintf(temp2, "%d", label[ltop - 3]);
  strcat(lab2, temp2);
  insert_quad("Label",NULL,NULL,lab2);
}

void for4() //go to autoincrement and print label after loop
{
    // printf("goto L%d \n",label[ltop - 2]);
    char lab[26] = "L";
    char temp1[100];
    sprintf(temp1, "%d", label[ltop - 2]);
    strcat(lab, temp1);
    insert_quad("goto",NULL,NULL,lab);

    // printf("L%d : \n",label[ltop - 1]);
    char lab2[26] = "L";
    char temp2[100];
    sprintf(temp2, "%d", label[ltop - 1]);
    strcat(lab2, temp2);
    insert_quad("Label",NULL,NULL,lab2);
    ltop = ltop - 4;
}
```

```
//if
void if1()
{
 lnum++;
 strcpy(temp,"t");
 strcat(temp,i_);
//  printf("%s = not %s %s %s\n",temp,st[top-3],st[top-2],st[top-1]);
 char temp1[50] = "not ";
 strcat(temp1, st[top-2]);
 insert_quad(temp1,st[top-3],st[top-1],temp);
 top -= 2;

//  printf("if %s goto L%d\n",temp,lnum);
 char lab2[26] = "L";
 char temp2[100];
 sprintf(temp2, "%d", lnum);
 strcat(lab2, temp2);
 insert_quad("if",temp,NULL,temp2);

 i_[0]++;
 label[++ltop]=lnum;
}
```

```
 void if2()
 {
 int x;
 lnum++;
 x=label[ltop--];

 char lab2[26] = "L";
 char temp2[100];
 sprintf(temp2, "%d", x);
 insert_quad("goto",NULL,NULL,temp2);
 // printf("goto L%d\n",lnum);

 // printf("L%d: \n",x);
 char lab[26] = "L";
 char temp1[100];
 sprintf(temp1, "%d", x);
 strcat(lab, temp1);

 insert_quad("Label",NULL,NULL,lab);
 label[++ltop]=lnum;
 }

 //if-else
 void if3()
 {
 int y;
 y=label[ltop--];
 //printf("L%d: \n",y);
 char lab[26] = "L";
 char temp1[100];
 sprintf(temp1, "%d", y);
 strcat(lab, temp1);

 insert_quad("Label",NULL,NULL,lab);

 }
```

ICG generation

```
=       10              a
 +      a       50      t00
 =      t00             b
 +      10.5    1       t01
 =      t01             z
 +      z       1       t02
 =      t02             z
 +      a       1       t03
 =      t03             a
 =      100             y
 +      a       50      t04
 =      t04             c
 *      100     100     t05
 +      a       t05     t06
 =      t06             e
param   a
param   b
param   c
call    foo     3
=       0               i
Label                   L1
not <   i       100     t07
if      t07             4
goto                    L2
Label                   L3
+       i       1       t08
=       t08             i
goto                    L1
Label                   L2
=       12              d
goto                    L3
Label                   L4
return 0
```

- Code Optimisation:
  1. Common Subexpression Elimination

```
=       10              a
+       a       50      t00
=       t00             b
+       10.5    1       t01
=       t01             z
+       z       1       t02
=       t02             z
+       a       1       t03
=       t03             a
=       100             y
=       t00             c
*       100     100     t05
+       a       t05     t06
=       t06             e
param   a
param   b
param   c
call    foo     3
=       0               i
Label                   L1
not <   i       100     t07
if      t07             4
goto                    L2
Label                   L3
+       i       1       t08
=       t08             i
Label                   L2
=       12              d
Label                   L4
return  0
```

2. Constant Propagation

```
=       10              a
+       10      50      t00
=       t00             b
+       10.5    1       t01
=       t01             z
+       z       1       t02
=       t02             z
+       10      1       t03
=       t03             a
=       100             y
=       t00             c
*       100     100     t05
+       10      t05     t06
=       t06             e
param   10
param   b
param   c
call    foo     3
=       0               i
Label                   L1
not <   0       100     t07
if      t07             4
goto                    L2
Label                   L3
+       0       1       t08
=       t08             i
Label                   L2
=       12              d
Label                   L4
return  0
```

3. Constant Folding

```
=       10              a
=       60              t00
=       t00             b
=       11.5            t01
=       t01             z
+       z       1       t02
=       t02             z
=       11              t03
=       t03             a
=       100             y
=       t00             c
=       10000           t05
+       10      t05     t06
=       t06             e
param   10
param   b
param   c
call    foo     3
=       0               i
Label                   L1
not <   0       100     t07
if      t07             4
goto                    L2
Label                   L3
=       1               t08
=       t08             i
Label                   L2
=       12              d
Label                   L4
return  0
```

4. Constant propagation

```
=       10              a
=       60              t00
=       60              b
=       11.5            t01
=       11.5            z
+       z       1       t02
=       t02             z
=       11              t03
=       11              a
=       100             y
=       60              c
=       10000           t05
+       10      10000   t06
=       t06             e
param   10
param   b
param   c
call    foo     3
=       0               i
Label                   L1
not <   0       100     t07
if      t07             4
goto                    L2
Label                   L3
=       1               t08
=       1               i
Label                   L2
=       12              d
Label                   L4
return  0
```

5. Dead code elimination and final optimisation:

```
=       60              b
=       60              c
param   10
param   b
param   c
call    foo     3
Label                   L1
not <   0       100     t07
if      t07             4
goto                    L2
Label                   L3
Label                   L2
Label                   L4
```

CONCLUSIONS

The compiler is capable of generating assembly code after checking for syntactic and semantic inconsistencies, for the core C++ concepts used for sequential execution, along with looping constructs such as for loops and conditional execution statements such as if and if-else statements.

FURTHER ENHANCEMENTS

- Code optimization could be improved further using more complex algorithms, so that the current methods can work on non sequential programs as well.
- A larger subset could be handled to yield a more powerful compiler.
- Error recovery and correction strategies could be implemented better to aid the programmer.
- Linked lists could be used for symbol table generation.

References

1. Design Compiler User Guide
2. 12 Design Compiler Interface - UCSD CSE
3. Synthesis Quick Reference - UCSD CSE