REPORT ON MINI COMPILER ON C++

SUBMITTED FOR THE REQUIREMENTS OF SEM VI

COMPILER DESIGN LABORATORY

BACHELOR IN TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY :

MAYANK AGARWAL	PES1201701349
R SIVA GIRISH	PES1201700159
PARSHVA B JAIN	PES1201701336

UNDER THE GUIDANCE OF
SUHAS GK
ASSISTANT PROFESSOR , PES UNIVERSITY.

Table Of Contents

INTRODUCTION

The objective of this project is to build a Mini C++ compiler using C programming language. The end product being generation of assembly level code which is previously code optimised using python programming language.

Since it is a Mini C++ compiler , it takes care of the majority of constructs in C++ language and the rest of the constructs are left for future enhancements.

It works for constructs which include loops such as **For** , **While** and constructs such as conditional statements such as **If** , **Else If**.
The various steps included in generating the optimised intermediate code includes :

- Generating symbol table after evaluating the expressions in the input file.
- Generate Abstract Syntax Tree for the code generated.
- Generate 3 address codes which are not optimised and are in the form of quadruples.
- Generate optimised code by performing basic code optimisation techniques.

The various tools used in building this Mini C++ compiler include LEX which helps in identification of predefined patterns and helped in generating tokens accordingly. YACC was used for parsing the input for semantic meaning and generating abstract syntax trees and intermediate code generation.

ARCHITECTURE OF LANGUAGE

The following C++ constructs have been implemented :

- While Loops
- Nested While Loops
- Simple If
- If Else
- For Loop
- If Else ladders

The following expressions are also being handled :

- We handled all kinds of Logical Expressions which include - Logical AND , Logical OR and Logical NOT.
- We handled all kinds of Arithmetic Expressions which include - **+ , - , * , / , ++ , --**
- We handled all kinds of Boolean Expressions which include - **> , < , <= , >= , == , != .**

Along with the various expressions included , Error reporting is also being included which includes :

- Error handling reports line numbers where error occured.
- Error handling reports the missing and the expected token in case of missing token.
- Error handling mechanism used for handling errors is “Panic Mode Recovery” and the synchronizing token used to identify it is “Semi colon ” i.e “ ; ” .

PHASE 1 : IMPLEMENTATION OF EACH PHASE OF COMPILER

PHASE 1.A : LEXICAL ANALYSIS

- In order to implement lexical analysis phase , we make use of LEX tool which is used to generate a lexical analyser .It does this by translating set of regular expressions specified in input.l file into a C implementation of a corresponding finite state machine i.e lex.yy.cc.
- So , we create a scanner using the LEX tool for C++ Language.This scanner transforms the source file from a stream of bytes and transforms into a series of predefined meaningful tokens.
- This scanner also takes care of removing both single and multiline comments and removes them for further analysis.
- All the tokens in the code are included in the form of T_<Token-Name> where <Token-Name> represents token name.
  For Example : T_multiply represents '*'.
- A Global variable called "yyval" which is used to record the value of lexeme scanned and a Global variable called "yytext" is the lex variable which stores the matched string.
- The lexical analyzer built also removes any whitespaces and comments along with breaking the syntax into a series of tokens.
  If the lexical analyzer recognizes that a token is invalid then it generates an error. The invalid token recognition task is achieved when the input string does not match any rule written in the lex file.
- The rules defined in the lex file eventually are regular expressions which also have corresponding actions associated with it which are executed on a match with the input stream.
- The lexical analyser built will eventually pass these tokens to syntax analyser whenever it demands.

Lex Rules:

```
\"[^\"]*\"                                    {printf("T_stringLiteral ");}
"'"(.|\\n)"'"                          {printf("T_character ");}
"//".*\n                {;}
"/*"[^*/]*"*/"                          {;}
```

```
alignas|alignof|and|and_eq|asm|atomic_cancel|atomic_commit|atomic_noexcept|bitand|bitor|brea
k|case|catch|class|compl|concept|const|consteval|constexpr|constinit|const_cast|continue|co_await|
co_return|co_yield|decltype|default|delete|do|dynamic_cast|enum|explicit|export|extern|false|frien
d|goto|inline|mutable|namespace|new|noexcept|not|not_eq|nullptr|operator|or|or_eq|private|protect
ed|public|reflexpr|register|reinterpret_cast|requires|signed|sizeof|static|static_assert|static_cast|stru
ct|switch|synchronized|template|this|thread_local|throw|true|try|typedef|typeid|typename|union|un
signed|using|virtual|void|volatile|wchar_t|xor|xor_e  {printf("T_keyword ");}

"if"
                        {printf("T_if  ");}
"else"
                        {printf("T_else  ");}

"int"
                        {printf("T_int  "); }
"main"
                        {printf("T_main  "); }
"auto"|"bool"|"char"|"char8_t"|"char16_t"|"char32_t"|"double"|"float"|"long"|"short"
{printf("T_type  ");}
"return"
                                {printf("T_return ");}

"for"                                           {printf("T_for ");}
"while"                                                 {printf("T_while ");}


"<<"                                            {printf("T_InputStream  ");}
">>"                                            {printf("T_OutputStream  ");}


"("                                                     {printf("T_openParenthesis  ");}

")"                                                     {printf("T_closedParanthesis  ");}
"{"
{count=count+1;top=top+1;st[top]=count;printf("T_openFlowerBracket  ");}
"}"
{top=top-1;printf("T_closedFlowerBracket  ");}

">"                                                     {printf("T_greater ");}
```

```
">="                                        {printf("T_greater_equal ");}
"<"                                                 {printf("T_less ");}
"<="                                        {printf("T_less_equal ");}
"=="                                        {printf("T_equal_equal ");}
"!="                                        {printf("T_not_equal ");}

"&&"                                        {printf("T_LogicalAnd ");}
"[|][|]"                            {printf("T_LogicalOr ");}



"+="|"-="|"*="|"/="|"%="            {printf("T_shortHand ");}

"+"                                                 {printf("T_plus ");}
"-"                                                 {printf("T_minus ");}
"/"                                                 {printf("T_divide ");}
"*"                                                 {printf("T_multiply ");}
"%"                                                 {printf("T_mod ");}
"="                                                 {printf("T_AssignmentOperator  ");}

";"                                                 {printf("T_Semicolon  ");}

{letter}({letter}|{digit})*     {Gen_Symbol_Table(yytext,yylineno,st[top]);printf("T_identifier
");  }

{digit}+                                    {printf("T_numericConstants ");}



[ \t]                   {/*printf("T_whiteSpace ");*/}
[\n]                                        {;}
.                                                   {printf("Invalid character found,
Abort!!!");}
```

PHASE 1.B : SYNTAX ANALYSIS

- The syntax analysis phase will analyse the syntactical structure of the input.Basically , this phase checks if the given input is in correct syntax in accordance with the programming language.
- It is responsible for verifying the sequence of tokens forms a valid sentence or not .It verifies this by making use of Programming Language Grammar defined.
  The project supports these implementations :
    - Variable declaration and initialization.
    - Variables of type : int , float , char are supported.
    - Arithmetic , Boolean and Logical expressions are supported.

- For the purpose of parsing , we use YACC which provides a tool to produce a parser for given grammar. YACC also reports shift-reduce and reduce-reduce conflicts generated on parsing an ambiguous grammar.YACC

CONTEXT FREE GRAMMAR USED :

```
Start : T_int T_main T_openParenthesis T_closedParanthesis openflower block_end_flower
{}


/* This production assumes flower bracket has been opened*/
block_end_flower : stmt Multiple_stmts
                                        | closeflower

/*This takes care of statements like if(...);. Note that to include multiple statements, a block has
to open with a flower bracket*/
block :  openflower block_end_flower
             | stmt
             | T_Semicolon
                   ;

/* block would cover anything following the statement. consider the for statement for example.
All possibilities are:
for(expr;expr;expr);
                  (block -> ;)
for(...) stmt         , where stmt contains T_Semicolon                                  (block ->
stmt)
```

```
for(...){}
                                        (block -> {block_end_flower -> {})
for(...){stmt, stmt, stmt, ...}
          (block -> {block_end_flower -> {smt Multiple_stmts})
for(...){stmt, if/while/for{stmt, stmt.}} , this is achieved implicitly because stmt in previous can
in turn be if or for while
*/


Multiple_stmts : stmt Multiple_stmts
                    |closeflower
                    ;

stmt : expr T_Semicolon                              {/*Statement cannot be empty, block
takes care of empty string*/}
                    | if_stmt
                    | while_stmt
                    | for_stmt
                    | Assignment_stmt T_Semicolon
                    | error T_Semicolon
                    ;


//for_stmt : T_for T_openParenthesis expr_with_semicolon expr_with_semicolon
expr_or_empty T_closedParanthesis block

for_stmt : T_for T_openParenthesis expr_or_empty_with_semicolon_and_assignment
expr_or_empty_with_semicolon_and_assignment
expr_or_empty_with_assignment_and_closed_parent  block

while_stmt : T_while T_openParenthesis expr T_closedParanthesis block

if_stmt : T_if T_openParenthesis expr T_closedParanthesis block elseif_else_empty

elseif_else_empty : T_else T_if T_openParenthesis expr T_closedParanthesis block
elseif_else_empty
                                                  | T_else Multiple_stmts_not_if
                                                  | T_else open flower block_end_flower
                                                  |
                                                  ;

Multiple_stmts_not_if : stmt_without_if Multiple_stmts
                                                  |T_Semicolon
                                                  ;

stmt_without_if : expr T_Semicolon
                                                  | Assignment_stmt T_Semicolon
                                                  | while_stmt
                                                  |for_stmt
                                                  ;

Assignment_stmt:                        T_identifier T_AssignmentOperator expr
                                                  | T_identifier T_shortHand expr
                                                  | T_type T_identifier T_AssignmentOperator
expr_without_constants   {insert_in_st($1, $2, st[top], "j");}
                                                  | T_type T_identifier T_AssignmentOperator
T_stringLiteral   {insert_in_st($1, $2, st[top], $4);}
                                                  | T_type T_identifier T_AssignmentOperator
T_numericConstants   {insert_in_st($1, $2, st[top], $4);}
                                                  | T_int T_identifier T_AssignmentOperator
expr_without_constants    {insert_in_st($1, $2, st[top], "j");}
                                                  | T_int T_identifier T_AssignmentOperator
T_numericConstants    {insert_in_st($1, $2, st[top], $4);}
                                        ;
expr_or_empty_with_semicolon_and_assignment: expr_or_empty T_Semicolon
          | Assignment_stmt T_Semicolon

expr_or_empty_with_assignment_and_closed_parent: expr_or_empty T_closedParanthesis
          | Assignment_stmt T_closedParanthesis

expr_without_constants:  T_identifier
                    | expr T_plus expr
                    | expr T_minus expr
                    | expr T_divide expr
                    | expr T_multiply expr
                    | expr T_mod expr
                    | expr T_LogicalAnd expr
                    | expr T_LogicalOr expr
```

```
              | expr T_less expr
              | expr T_less_equal expr
              | expr T_greater expr
              | expr T_greater_equal expr
              | expr T_equal_equal expr
              | expr T_not_equal expr
              ;


expr:   T_numericConstants
              | T_stringLiteral
              | T_identifier
              | expr T_plus expr
              | expr T_minus expr
              | expr T_divide expr
              | expr T_multiply expr
              | expr T_mod expr
              | expr T_LogicalAnd expr
              | expr T_LogicalOr expr
              | expr T_less expr
              | expr T_less_equal expr
              | expr T_greater expr
              | expr T_greater_equal expr
              | expr T_equal_equal expr
              | expr T_not_equal expr
              ;

expr_or_empty: expr
                     |
                     ;

openflower: T_openFlowerBracket {};
closeflower: T_closedFlowerBracket {};
```

PHASE 2 : GENERATION OF SYMBOL TABLE ALONG WITH EXPRESSION EVALUATION

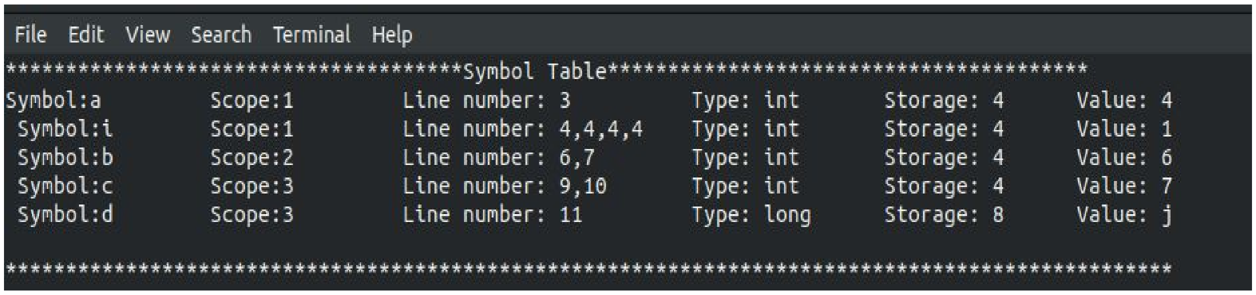
- For the generation of symbol tables , we maintain a structure which keeps track of all necessary objects which is further required for analysis.
- The structure of the symbol table contains variable name , Line number , type , value , scope. The structure is as follows :

Structure:

```
struct var
{
        char var_name[20];
        char Line_t[100];
        // char type[100];
        int scope;
};
struct symbol_table_entry
{
        struct var arr[20];
        int up;
};
```

- As each line is parsed , the actions associated with grammar rules are executed.
- “ $1 ” is used to represent the first token in the production and “ $$ ” is used to represent the resultant of the given production.
- When the parsing is done , the generated symbol table is printed on the console.

SNAPSHOT OF SYMBOL TABLE :



```
*************************************Symbol Table****************************************
Symbol:a        Scope:1        Line number: 3          Type: int       Storage: 4      Value: 4
 Symbol:i       Scope:1        Line number: 4,4,4,4    Type: int       Storage: 4      Value: 1
 Symbol:b       Scope:2        Line number: 6,7        Type: int       Storage: 4      Value: 6
 Symbol:c       Scope:3        Line number: 9,10       Type: int       Storage: 4      Value: 7
 Symbol:d       Scope:3        Line number: 11         Type: long      Storage: 8      Value: j

***********************************************************************************************
```

PHASE 3 : GENERATION OF ABSTRACT SYNTAX TREE

- For the generation of abstract syntax trees , a tree structure which represents the syntactical flow of the code is created.
- For expressions , associativity is indicated using %left and %right fields.
- For the case of precedence of operations , the last rule gets higher precedence. For example :

```
%left T_LogicalAnd T_LogicalOr
%left T_less T_less_equal T_greater T_greater_equal T_equal_equal T_not_equal
%left T_plus T_minus
%left T_multiply T_divide T_mod
```

- To build a tree , a structure is maintained which has 2 pointers which points to its children and token field for storage.

```
typedef struct ASTNode
{
        struct ASTNode *left;
        struct ASTNode *right;
        char *token;
} node;
```

- When every new token is encountered during parsing , the function takes the value of the token and then creates a node for the tree and then makes sure that it attaches to its parent.When the head production of the construct is reached , we display the node.

Sample input:

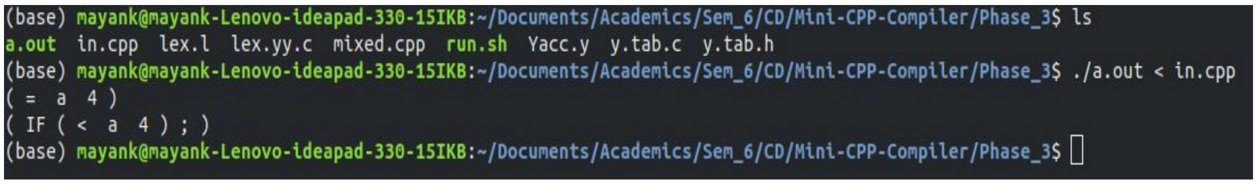
```
 int main()
{
        int a = 4;
        if(a < 4)
        {

        }
}
```

Sample output:

```
( =  a  4 )
( IF ( <  a  4 ) ; )
```

SNAPSHOT OF ABSTRACT SYNTAX TREE GENERATED :

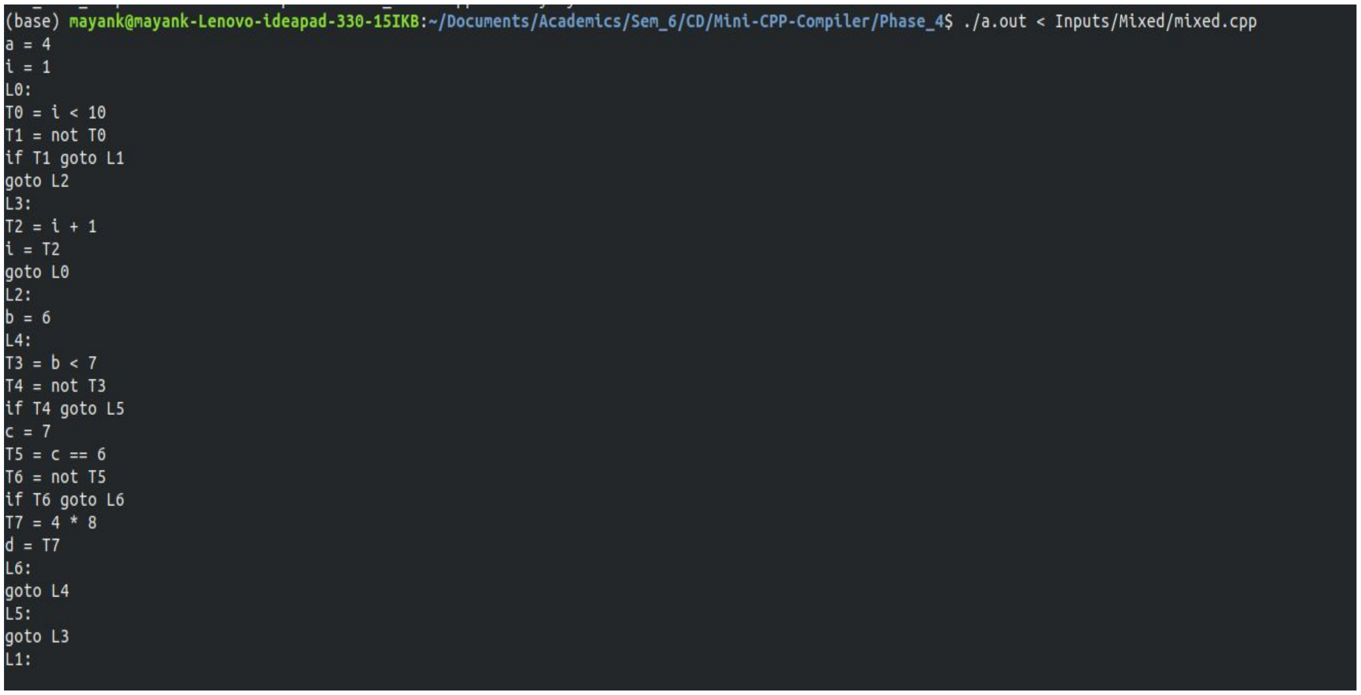


```
(base) mayank@mayank-Lenovo-ideapad-330-15IKB:~/Documents/Academics/Sem_6/CD/Mini-CPP-Compiler/Phase_3$ ls
a.out  in.cpp  lex.l  lex.yy.c  mixed.cpp  run.sh  Yacc.y  y.tab.c  y.tab.h
(base) mayank@mayank-Lenovo-ideapad-330-15IKB:~/Documents/Academics/Sem_6/CD/Mini-CPP-Compiler/Phase_3$ ./a.out < in.cpp
( =  a  4 )
( IF ( <  a  4 ) ; )
(base) mayank@mayank-Lenovo-ideapad-330-15IKB:~/Documents/Academics/Sem_6/CD/Mini-CPP-Compiler/Phase_3$
```

PHASE 4 : INTERMEDIATE CODE GENERATION

- Intermediate code generation phase takes its input from semantic analyzer phase in the form of annotated syntax tree.This syntax tree is then converted into linear representation.
- Three Address Code : A statement which involves no more than 3 references (two for operands and one for result).A sequence of three address statements is known as three address code.
- It is of the form : a = b op c where a,b,c will have memory location.
  Example : The three address code for the expression : x * y + z will be :
  ```
  T1 = x * y
  T2 = T1 + z
  T3 = T2
  ```
  Where T1,T2,T3 are temporary variables.
- The data structure used to represent three address codes is Quadruples which is a structure in C programming language. It has 4 columns namely : operator , operand1 ,operand2 and result.
- Due to the usage of Quadruples , our code may contain a lot of temporary variables which can increase time and space complexity.
- On the other hand , usage of Quadruples makes it easy to rearrange code for global optimization.

SNAPSHOTS OF INTERMEDIATE CODE GENERATION



```
(base) mayank@mayank-Lenovo-ideapad-330-15IKB:~/Documents/Academics/Sem_6/CD/Mini-CPP-Compiler/Phase_4$ ./a.out < Inputs/Mixed/mixed.cpp
a = 4
i = 1
L0:
T0 = i < 10
T1 = not T0
if T1 goto L1
goto L2
L3:
T2 = i + 1
i = T2
goto L0
L2:
b = 6
L4:
T3 = b < 7
T4 = not T3
if T4 goto L5
c = 7
T5 = c == 6
T6 = not T5
if T6 goto L6
T7 = 4 * 8
d = T7
L6:
goto L4
L5:
goto L3
L1:
```

PHASE 5 : Optimization

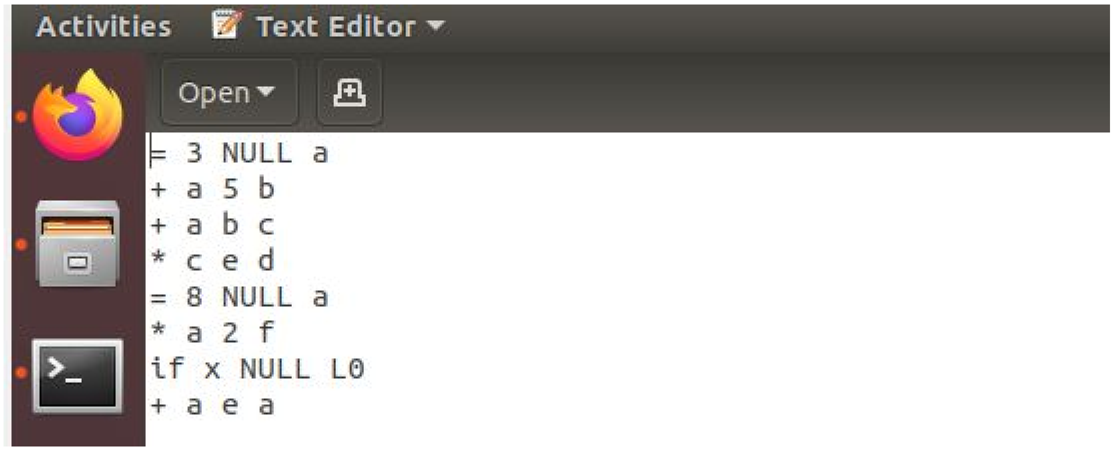
- The optimization method used is constant folding , constant propagation followed by dead code elimination. This uses the fact that the compiler can implicitly do some computations.

- Constant Folding : Expressions having constant operands can be evaluated at run time and thereby increasing the performance of the code.

- Constant Propagation : Here , we substitute the values of known variables in the expressions which enables the code to assign static values which is better and faster than looking up and copying values of variables in the register and hence increasing the performance of the code.

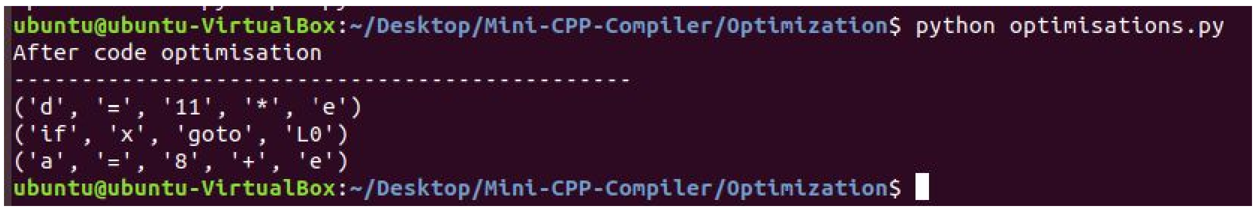
- By performing constant folding and constant propagation , dead code elimination is also taken care of to an extent.

SNAPSHOT OF CODE OPTIMISATION :

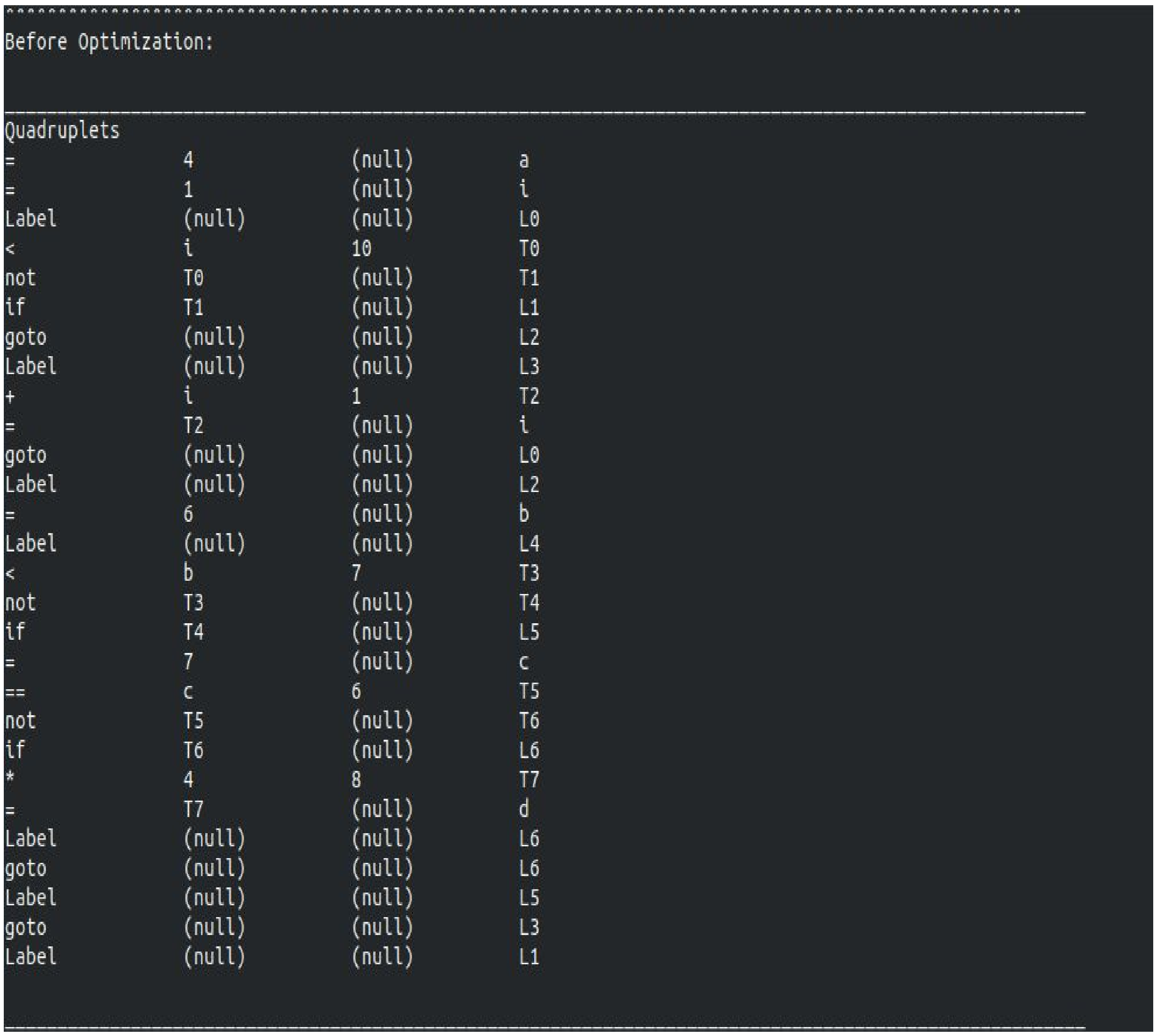


```
= 3 NULL a
+ a 5 b
+ a b c
* c e d
= 8 NULL a
* a 2 f
if x NULL L0
+ a e a
```

SNAPSHOT OF CODE AFTER OPTIMISATION


```
ubuntu@ubuntu-VirtualBox:~/Desktop/Mini-CPP-Compiler/Optimization$ python optimisations.py
After code optimisation
---------------------------------------------
('d', '=', '11', '*', 'e')
('if', 'x', 'goto', 'L0')
('a', '=', '8', '+', 'e')
ubuntu@ubuntu-VirtualBox:~/Desktop/Mini-CPP-Compiler/Optimization$
```

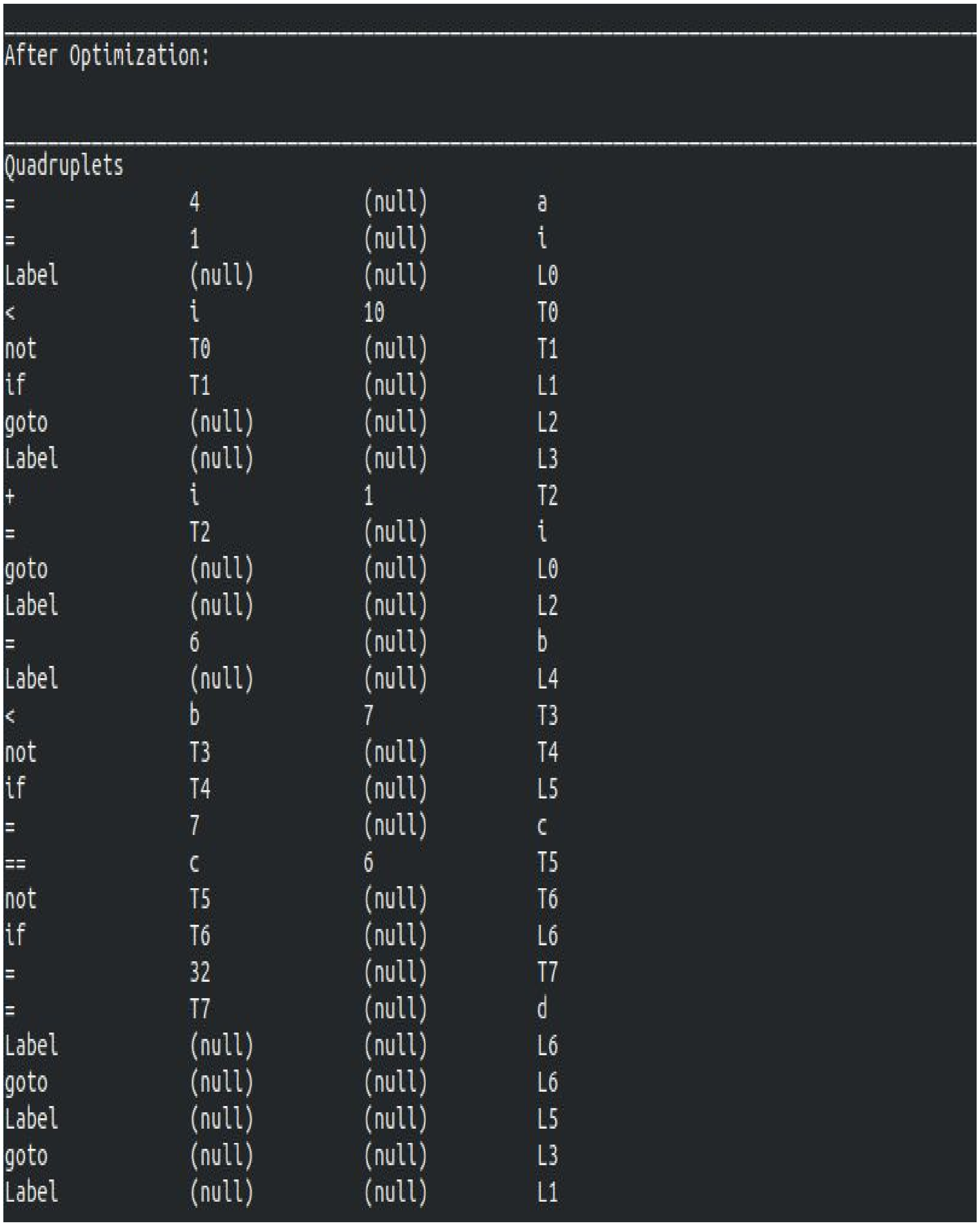
SNAPSHOT OF CODE BEFORE OPTIMISATION


```
Before Optimization:


----------------------------------------------------------------------------------------------
Quadruplets
=               4               (null)          a
=               1               (null)          i
Label           (null)          (null)          L0
<               i               10              T0
not             T0              (null)          T1
if              T1              (null)          L1
goto            (null)          (null)          L2
Label           (null)          (null)          L3
+               i               1               T2
=               T2              (null)          i
goto            (null)          (null)          L0
Label           (null)          (null)          L2
=               6               (null)          b
Label           (null)          (null)          L4
<               b               7               T3
not             T3              (null)          T4
if              T4              (null)          L5
=               7               (null)          c
==              c               6               T5
not             T5              (null)          T6
if              T6              (null)          L6
*               4               8               T7
=               T7              (null)          d
Label           (null)          (null)          L6
goto            (null)          (null)          L6
Label           (null)          (null)          L5
goto            (null)          (null)          L3
Label           (null)          (null)          L1


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

SNAPSHOT OF CODE AFTER OPTIMISATION



```
After Optimization:


Quadruplets
=           4           (null)      a
=           1           (null)      i
Label       (null)      (null)      L0
<           i           10          T0
not         T0          (null)      T1
if          T1          (null)      L1
goto        (null)      (null)      L2
Label       (null)      (null)      L3
+           i           1           T2
=           T2          (null)      i
goto        (null)      (null)      L0
Label       (null)      (null)      L2
=           6           (null)      b
Label       (null)      (null)      L4
<           b           7           T3
not         T3          (null)      T4
if          T4          (null)      L5
=           7           (null)      c
==          c           6           T5
not         T5          (null)      T6
if          T6          (null)      L6
=           32          (null)      T7
=           T7          (null)      d
Label       (null)      (null)      L6
goto        (null)      (null)      L6
Label       (null)      (null)      L5
goto        (null)      (null)      L3
Label       (null)      (null)      L1
```

ASSEMBLY CODE GENERATION

The final phase of the compiler design is assembly code generation which takes input as intermediate code generated along with the symbol table and then outputs target assembly code.

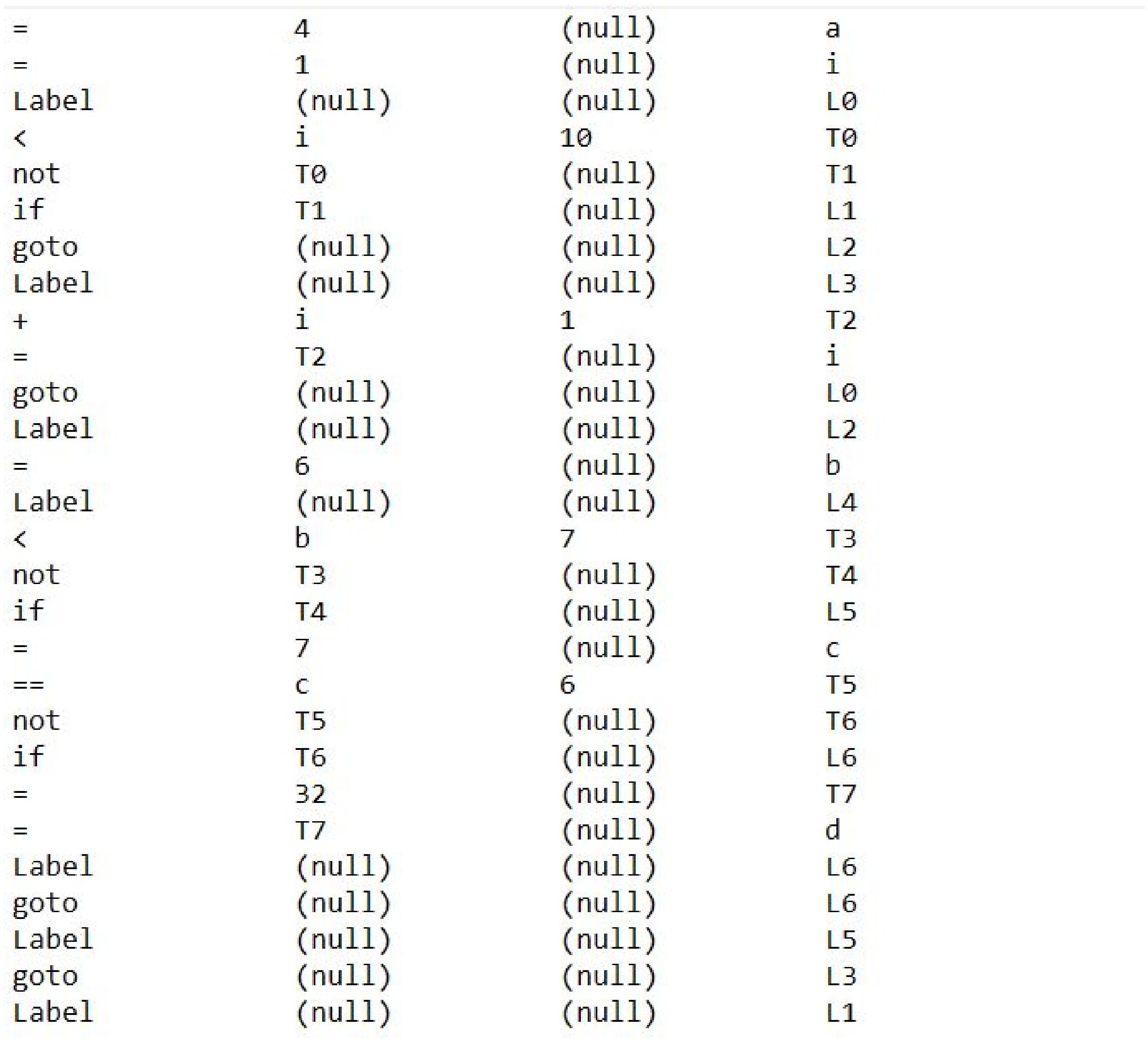
- This phase proceeds on the assumption that the input it gets is free of any kind of errors including syntax and semantic errors.
- Allowed operations to be used for code generation include **MOV , LD , ST , ADD , SUB , CMP , BLT , BGE , BLE , BGT ,BEQ , BNE** and many more.
- Some of the design issues we face during code generation which includes Register allocation strategy , Choice of evaluation order and Instruction selection.
- For the sake of our project , the number of registers available for disposal is 16.
- While converting code from intermediate code generation to assembly code , it is also necessary to preserve the semantics of the source program.
- Assembly code generated can directly be converted to binary form i.e 0's and 1's by making use of opcode.
- During this phase , we also need to make sure that we optimally use the available registers since we only have limited registers available.

The techniques employed for optimally using available registers :

- If registers are not going to be used again in the current block , then we clear the register.
- If we encounter block shifts , then we release registers.

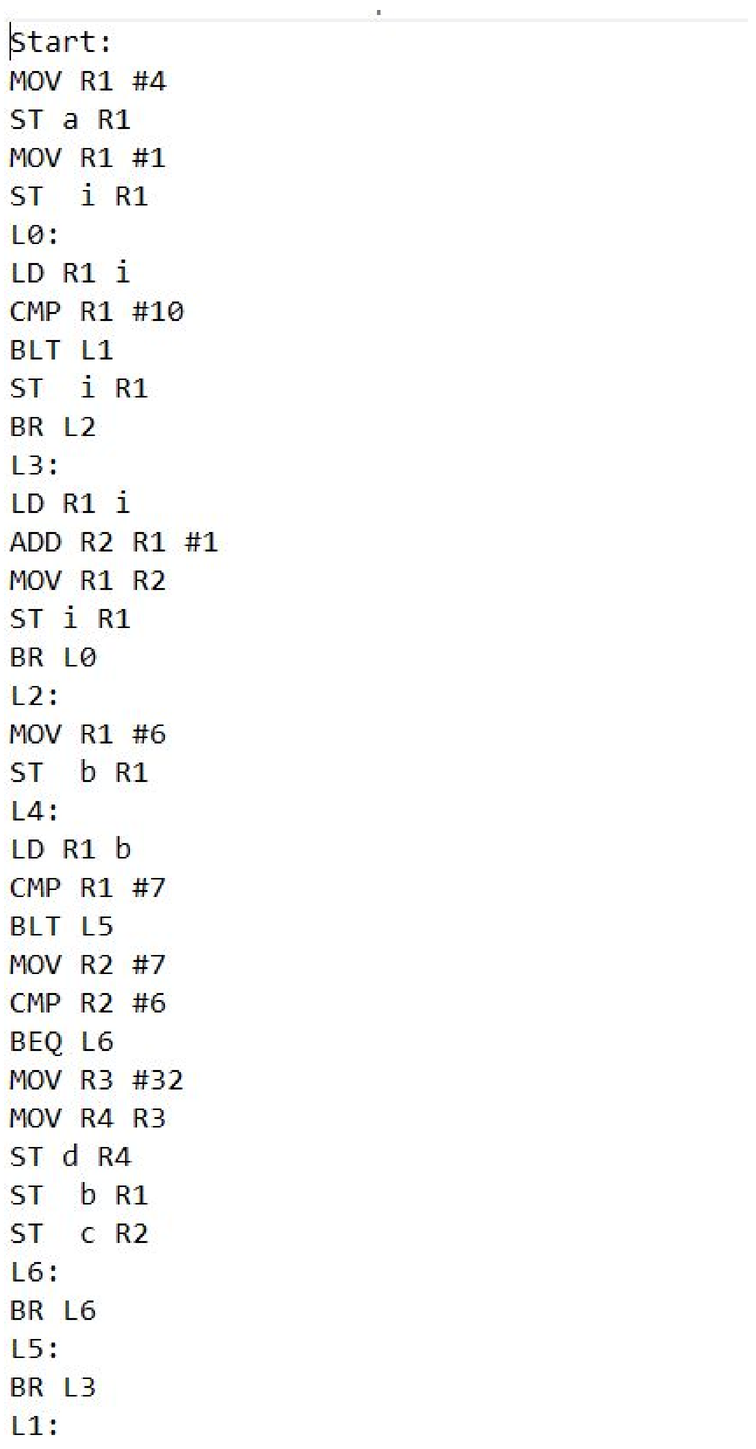
SNAPSHOT OF ASSEMBLY CODE GENERATOR INPUT

INPUT



```
=               4               (null)          a
=               1               (null)          i
Label           (null)          (null)          L0
<               i               10              T0
not             T0              (null)          T1
if              T1              (null)          L1
goto            (null)          (null)          L2
Label           (null)          (null)          L3
+               i               1               T2
=               T2              (null)          i
goto            (null)          (null)          L0
Label           (null)          (null)          L2
=               6               (null)          b
Label           (null)          (null)          L4
<               b               7               T3
not             T3              (null)          T4
if              T4              (null)          L5
=               7               (null)          c
==              c               6               T5
not             T5              (null)          T6
if              T6              (null)          L6
=               32              (null)          T7
=               T7              (null)          d
Label           (null)          (null)          L6
goto            (null)          (null)          L6
Label           (null)          (null)          L5
goto            (null)          (null)          L3
Label           (null)          (null)          L1
```

OUTPUT



```
Start:
MOV R1 #4
ST a R1
MOV R1 #1
ST  i R1
L0:
LD R1 i
CMP R1 #10
BLT L1
ST  i R1
BR L2
L3:
LD R1 i
ADD R2 R1 #1
MOV R1 R2
ST i R1
BR L0
L2:
MOV R1 #6
ST  b R1
L4:
LD R1 b
CMP R1 #7
BLT L5
MOV R2 #7
CMP R2 #6
BEQ L6
MOV R3 #32
MOV R4 R3
ST d R4
ST  b R1
ST  c R2
L6:
BR L6
L5:
BR L3
L1:
```

References

1. Hyperlinked C++ BNF Grammar
2. Compiler - Intermediate Code Generation
3. Intermediate Code Generation in Compiler Design
4. Abstract syntax tree
5. Symbol Table in Compiler
6. Compiler Design - Symbol Table
7. Compiler Design - Code Optimization
8. Constant folding