Report on

Mini C++ Compiler

Submitted in partial fulfilment of the requirements for ***Sem VI.***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Vishnu S Reddy	**PES2201800118**
Keshav Raju R	**PES2201800135**
Ishan Padhy	**PES2201800158**

Under the guidance of

Prof. Swati Gambhire
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

Chapter No.	Title	Page No.
1.	INTRODUCTION	03
2.	ARCHITECTURE OF LANGUAGE	04
3.	LITERATURE SURVEY	04
4.	CONTEXT FREE GRAMMAR	05
5.	DESIGN STRATEGY ● SYMBOL TABLE CREATION ● INTERMEDIATE CODE GENERATION ● CODE OPTIMIZATION ● ERROR HANDLING	10
6.	IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following): ● SYMBOL TABLE CREATION ● INTERMEDIATE CODE GENERATION ● CODE OPTIMIZATION ● ERROR HANDLING	10
7.	RESULTS	13
8.	SNAPSHOTS	13
9.	CONCLUSIONS	20
10.	FURTHER ENHANCEMENTS	20
REFERENCES/BIBLIOGRAPHY		20

1. INTRODUCTION

The aim of this project is to implement a mini compiler for C++ programs. This compiler should be able to generate the optimized intermediate code from the given C++ source code. This compiler will be able to handle if-else and loop constructs of C++.

This compiler also detects various errors and reports them to user so that he can make the necessary changes in the source code. We have made use of Flex (Lex) and Bison (Yacc) along with the GNU GCC compiler to implement this mini-C++ compiler. A sample input and output are shown below.

```
#include<iostream>
int main(){
    int a = 1;
    int b = 1;
    //This is a loop
    int i = 0;
    while(i < 10){
        i = i+1;
    }
    //This is an if statement
    if (i == 10){
        a = 10;
    }
    else{
        b = 10;
    }
     int c = a + b;
}
```

Figure 1. This is the input C++ program.

```
a = 1
b = 1
i = 0
L0 :
IF i<10 GOTO L1
GOTO L2
L1 :
i = t0
GOTO L0
L2 :
IF i==10 GOTO L3
GOTO L4
L3 :
a = 10
GOTO L5
L4 :
b = 10
L5 :
c = 20
```

Figure 2. This is the sample output to the above code.

2. ARCHITECTURE OF LANGUAGE

The architecture and syntax are very similar to C++, for all the constructs that we are focussing on. The semantics are very close to ISO C++. We have also handled the most common errors like using variables that are not declared, semicolon missing, bracket missing and many other such errors. We do not stop parsing as soon as an error is found, but instead we carry on and, in the end, print all the errors that were encountered. The data types, comments, keywords, and other constructs are mentioned below.

- Data Types – int, char, float, double.
- Comments – Single Line Comments //, Multi Line Comments /* */.
- White Space – Ignored and Tab Space converted to White Space.
- Keywords – main, if, else if, else, while, class, include.
- Looping Constructs Supported – while and for loops.
- If Else Constructs Supported – if, else if and else.
- Operators - +, *, /, -, &, <, <=, ==, >, >= and |.
- Delimiters - ; and ,.

3. LITERATURE SURVEY

The books, documents and online tutorials referred are listed here for your reference.

- Lex & Yacc, O’Reilly (http://index-of.co.uk/Misc/O'Reilly%20Lex%20and%20Yacc.pdf)
- The Lex & Yacc Page (http://dinosaur.compilertools.net/)
- Lex and Yacc Tutorial by Tom Niemann (https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf)

4. CONTEXT FREE GRAMMAR

We make use of BNF (Backus-Normal Form) of the context free grammar to develop this mini compiler. Most of these productions are inspired by ISO C++ form and we have created a few constructs to provide some novelty to our project. It is provided below.

preprocessing-token:

    #include header-name
    #define literal literal

declaration-statement:

    attribute declaration-specifier

function-definition:

    keyword identifier (identifier-sequence) function-body

function-body:

    compound-statement

token:

    identifier
    keyword
    literal
    operator-token
    punctuator

header-name:

    < string-path >
    " string-path "

identifier:

    identifier-nondigit
    identifier digit

statement:

    labeled-statement
    expression-statement
    compound-statement
    selection-statement

iteration-statement
jump-statement

labeled-statement
identifier : statement

compound-statement:
{ statement-seq }

statement-seq:
statement
statement-seq statement

selection-statement:
if (condition) statement
if (condition) statement else statement

condition:
expression
type-specifier-seq declarator = assignment-expression

iteration-statement:
while (condition) statement
for (for-range-declaration : for-range-initializer) statement

for-range-declaration:
attribute-specifier-seqopt type-specifier-seq declarator

for-range-initializer:
expression braced-init-list

jump-statement:
break ;
continue ;
return expressionopt ;
return braced-init-listopt ;

expression:
multiplicative-expression
additive-expression
relational-expression
equality-expression
logical-and-expression
logical-or-expression

conditional-expression
assignment-expression

multiplicative-expression:
pm-expression
multiplicative-expression * pm-expression
multiplicative-expression / pm-expression
multiplicative-expression % pm-expression

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

relational-expression:
shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

logical-and-expression:
Inclusive-or-expression
logical-and-expression && inclusive-or-expression

logical-or-expression:
logical-and-expression
logical-or-expression || logical-and-expression

conditional-expression:
logical-or-expression
logical-or-expression ? expression : assignment-expression

assignment-expression:
Conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression

assignment-operator:
=
*=

/=
%=
+=
-=
>>=
<<=
&=
^=
|=

identifier-nondigit:
nondigit

nondigit:
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z

id-expression:
identifier

Identifier-digit:
0

1
2
3
4
5
6
7
8
9

keyword:
bool
break
char
continue
double
else
false
float
for
if
int
long
return
true
void

punctuator:
{
}
[
]
#
(
)
;
:
?

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal

integer-literal:
    decimal-literal integer-suffixopt

decimal-literal:
    nonzero-digit
    decimal-literal digit

nonzero-digit:
    1
    2
    3
    4
    5
    6
    7
    8
    9

c-char:
    any member of the source character set except the single quote ', backslash \, or new-line character
    escape-sequence
    universal-character-name

escape-sequence:
    simple-escape-sequence

simple-escape-sequence:
    \'
    \"
    \\
    \n
    \t

sign:
    +
    -

digit-sequence:
    digit
    digit-sequence digit

string-literal:
    s-char-sequence

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except the double-quote ", backslash \, or new-line character

escape-sequence

universal-character-name

boolean-literal:

false

true

5. DESIGN STRATEGY

- **SYMBOL TABLE GENERATION**

  We create a symbol table that shows the Line, the name, the scope, the value, the id and the data type for each entry in the symbol table. We have managed to detect errors in this phase as well. It can identify syntax errors like missing semicolon and if we try to use a variable that is not declared. It can also detect re-declaration of variables.

- **INTERMEDIATE CODE GENERATION**

  Intermediate code generator receives the code from the previous phases of this compiler, which includes the lexical analyser, syntax analyser and the semantic analyser. The input to this phase is as a syntax tree that can then be pre-formatted to convert it into a linear representation. Intermediate code that is generated is machine independent code.

- **CODE OPTIMIZATION**

  We perform multiple code optimization techniques on the intermediate code generated in the previous phase to reduce the number of instructions and effectively speed up the process of execution.

- **ERROR HANDLING**

  In this compiler we check for syntactic and semantic errors like symbol missing, invalid syntax and undeclared identifiers (variables).

6. IMPLEMENTATION DETAILS

- **SYMBOL TABLE GENERATION**

  We use a linked list data structure to implement our symbol table. The print function outputs the formatted symbol table to STDOUT.

  Each node from the symbol table has the following structure.

  - **Line**

  - **Name**
  - **Scope**
  - **Value**
  - **ID**
  - **Data Type**

  These are displayed in a tabular form during the end of the program to represent the symbol table that has been generated.

- **INTERMEDIATE CODE GENERATION**
  We make use of three address code for representing the intermediate source code. Three address code is a type of intermediate code which is east to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler.

  The three-address code is given by the following general representation **– a = b op c**, where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator.

  We also must output the three-address code in the quadruple format. Quadruples have four fields to implement the three-address code. The field of the quadruple contains the name of the operator, the first source operand, the second source operand and the result, respectively.

Operator	Source 1	Source 2	Destination

- **CODE OPTIMIZATION**
  For code optimization we have performed four basic tasks that we were expected to perform as per the requirements of this project.

  - **LIVE VARIABLE ANALYSIS**
    Live variable analysis is a classic data-flow analysis to calculate the variables that are live at each point in the program. A variable is live at some point if it holds a value that may be needed in the future or equivalently if its value may be read before the next time the variable is written to.

  - **DEAD CODE ELIMINATION**
    In compiler theory, dead code elimination also known as DCE is a compiler optimization to remove dead code – code which does not affect the program results. It can also enable further optimizations by simplifying the program structure.

  - **LOOP INVARIANT CODE MOTION**
    Loop Invariant Code Motion is a well-known ACET optimization. It

recognizes these computations within a loop that produce the same result each time the loop is executed. These computations are called loop-invariant code and can eb moved outside the loop body without changing the program semantics.

- **CONSTANT FOLDING AND PROPOGATION**
  Expressions having constant operands can be evaluated at run time and thereby increasing the performance of the code.

  Here, we substitute the values of the 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.

- **ERROR HANDLING**
  Our mini compiler can handle quite a few common errors. These include the ones listed below.
  - **SYNTAX ERRORS**
    Syntax errors are the errors that are in the source code of a program. These are the errors that are caused by not following the syntax of the program properly. If any line or block of code is not following the code properly, it will raise an error, but it does not stop parsing. It also prints the total number of errors in the end.

  - **UNDECLARED VARIABLES**
    Our compiler can handle undeclared variables. This means that we use a variable/identifier without declaring or initializing it. This variable is not allocated memory and so it raises an error.

  - **REDECLARED VARIABLES**
    If a variable is declared multiple times, then it raises an error. This means if our code has declared a variable like int a = 10; and then again, we use int a anywhere in the code, then it identifies such errors and reports it.

INSTRUCTIONS TO RUN THE CODE:
All instructions required to run and reproduce the results is provided in our GitHub repository available at - https://github.com/vishnureddys/mini-cpp-compiler.

7. RESULTS

The compiler that we have designed performs all the required tasks as instructed in the project guidelines document. This includes pre-processing, lexical analysis, token generation, symbol table generation, parsing, intermediate code generation and intermediate code optimization.

8. SNAPSHOTS

GENERATION OF SYMBOL TABLE AND IDENTIFICATION OF ERRORS

```
C:\Users\vishn\Projects\mini-cpp-compiler\Phase 1>a < input.cpp



Symbol Table

   Symbol                   Name       Type         Scope        Line Number         Value
------------------------------------------------------------------------------------------
 identifier                    v         int          0              2              100
   function                 temp         int          0              3              0
 identifier                    j         int          1              5              10
        ╪"i                 main         int          0              8              0
 identifier                    x         int          1              13             0
 identifier                    p         int          1              13             5
 identifier                    a         int          1              14             10
 identifier                    c         int          1              15             10
 identifier                    i         int          1              17             1
 identifier                    j         int          2              20             7
```

Figure 3. When there is no error in the code.

```
C:\Users\vishn\Projects\mini-cpp-compiler\Phase 1>a < input.cpp

Line:15: error: use of undeclared identifier 'c'



Symbol Table

   Symbol                   Name       Type         Scope        Line Number         Value
------------------------------------------------------------------------------------------
 identifier                    v         int          0              2              100
   function                 temp         int          0              3              0
 identifier                    j         int          1              5              10
        h#╙                 main         int          0              8              0
 identifier                    x         int          1              13             0
 identifier                    p         int          1              13             5
 identifier                    a         int          1              14             10
 identifier                    i         int          1              17             1
 identifier                    j         int          2              20             7
```

Figure 4. When there is an error in the code.

We can see that the generation of the symbol table does not stop after detecting an error. It goes on and reports the errors only at the end.

INTERMEDIATE CODE GENERATION

```
Intermediate Code Generation (Quadraple Form):


---------------------------------------------------------
=        3        (null)   a
+        5        a        T0
=        T0       (null)   b
+        a        b        T1
=        T1       (null)   c
=        T1       (null)   d
=        8        (null)   a
*        2        a        T2
=        T2       (null)   f
=        10       (null)   d
not      b        (null)   T3
if       T3       (null)   L0
+        a        2        T4
=        T4       (null)   a
Label    (null)   (null)   L0


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

Figure 5. The intermediate code generated in Quadruple Form.

```
C:\Users\vishn\Projects\mini-cpp-compiler\Phase 2\Optimization>a input.cpp
iostream
a =  1
b =  1
i =  0
L0 :
if  i<10  goto L1
goto L2
L1 :
t0 = i + 1
i =  t0
goto L0
L2 :
if  i==10  goto L3
goto L4
L3 :
a =  10
goto L5
L4 :
b =  10
L5 :
t1 = a + b
c =  t1
```

Figure 6. ICG in three address code.

Line	Name	Scope	value	id_type	datatype
3	a	-1	10	IDENT	int
4	b	-1	10	IDENT	int
6	i	-1	t0	IDENT	int
8	t0	-1	i+1	TEMP	TEMP
17	t1	-1	a+b	TEMP	TEMP
17	c	-1	t1	IDENT	int
18	main	0		FUNCT	int

Figure 7. This is the symbol table for the ICG.

```
Generated ICG
a =  1
b =  1
i =  0
L0 :
if  i<10  goto L1
goto L2
L1 :
t0 = i + 1
i =  t0
goto L0
L2 :
if  i==10  goto L3
goto L4
L3 :
a =  10
goto L5
L4 :
b =  10
L5 :
t1 = a + b
c =  t1
```

Figure 8. This is the generated ICG that is provided as input to the Optimization Phase.

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

Live Variable Analysis

----------------------
Live Variables at Line  1  Are :  ['a']
Live Variables at Line  2  Are :  ['a', 'b']
Live Variables at Line  3  Are :  ['a', 'b', 'i']
Live Variables at Line  4  Are :  ['a', 'b', 'i']
Live Variables at Line  5  Are :  ['a', 'b', 'i']
Live Variables at Line  6  Are :  ['a', 'b', 'i']
Live Variables at Line  7  Are :  ['a', 'b', 'i']
Live Variables at Line  8  Are :  ['a', 'b', 'i', 't0']
Live Variables at Line  9  Are :  ['a', 'b', 'i']
Live Variables at Line  10  Are :  ['a', 'b', 'i']
Live Variables at Line  11  Are :  ['a', 'b', 'i']
Live Variables at Line  12  Are :  ['a', 'b', 'i']
Live Variables at Line  13  Are :  ['a', 'b', 'i']
Live Variables at Line  14  Are :  ['a', 'b', 'i']
Live Variables at Line  15  Are :  ['a', 'b', 'i']
Live Variables at Line  16  Are :  ['a', 'b', 'i']
Live Variables at Line  17  Are :  ['a', 'b', 'i']
Live Variables at Line  18  Are :  ['a', 'b', 'i']
Live Variables at Line  19  Are :  ['a', 'b', 'i']
Live Variables at Line  20  Are :  ['a', 'i', 't1']
Live Variables at Line  21  Are :  ['a', 'i']
```

Figure 9. We then perform Live Variable Analysis on this.

We then make us of these in the next stage that is in the loop invariant analysis. This involves removing repetitive statements that inside a loop. For example, in the above example, a = 10; is inside the loop. This can be removed, and this would help in increasing the speed of the compiler.

```
--------------------------
Loop Invariant Code Motion
--------------------------
a =  1
b =  1
i =  0
L0 :
if  i<10  goto L1
goto L2
L1 :
t0 = i + 1
i =  t0
goto L0
L2 :
if  i==10  goto L3
goto L4
L3 :
a =  10
goto L5
L4 :
b =  10
L5 :
t1 = a + b
c =  t1
```

Figure 10. IC after performing loop invariant code motion.

```
---------------------
Dead Code Elimination
---------------------
a =  1
b =  1
i =  0
L0 :
if  i<10  goto L1
goto L2
L1 :
t0 = i + 1
i =  t0
goto L0
L2 :
if  i==10  goto L3
goto L4
L3 :
a =  10
goto L5
L4 :
b =  10
L5 :
t1 = a + b
c =  t1
```

Figure 11. After performing Dead Code Elimination.

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

Constant Folding Quadruples

---------------------------

= 1 NULL a
= 1 NULL b
= 0 NULL i
LABEL L0
IF i<10 GOTO L1
GOTO L2
LABEL L1
+ i 1 t0
= 0 NULL t0
GOTO L0
LABEL L2
IF i==10 GOTO L3
GOTO L4
LABEL L3
= 10 NULL a
GOTO L5
LABEL L4
= 10 NULL b
LABEL L5
= 20 NULL t1
= 20 NULL c
```

Figure 12. Quadruples after performing constant folding and propagation.

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

Constant Folded Expression

--------------------------

a = 1
b = 1
i = 0
L0 :
IF i<10 GOTO L1
GOTO L2
L1 :
i = t0
GOTO L0
L2 :
IF i==10 GOTO L3
GOTO L4
L3 :
a = 10
GOTO L5
L4 :
b = 10
L5 :
c = 20
```

Figure 13. The optimized intermediate code.

9. CONCLUSION

With this project, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler which generates an intermediate code, given a C++ source code as input. Apart from this our compiler also identifies and reports errors that are present in the source code.

10. FUTHER ENHANCEMENTS

The current compiler can only handle if statements, loops. This can be improved to handle switch case statements, perform exception handling and a lot more. We should also be able to import external libraries. Another important extension to this would be to generate the assembly code from the generated intermediate optimized code.

11. REFERENCES/BIBLIOGRAPHY

The books, documents and online tutorials referred are listed here for your reference.

- Lex & Yacc, O'Reilly (http://index-of.co.uk/Misc/O'Reilly%20Lex%20and%20Yacc.pdf)
- The Lex & Yacc Page (http://dinosaur.compilertools.net/)
- Lex and Yacc Tutorial by Tom Niemann (https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf)