Report on

"C++ Compiler Using Lex and Yacc"

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Vansh angaria	**122cs0010**
Krishnamurthi	**122cs0067**

Under the guidance of

Dr K Nagaraju
Assistant Professor
IIITDM,Kurnool

`

TABLE OF CONTENTS

Introduction

C++ is one of the world's most popular programming languages.C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

The objective of our project was to create a simple C++ compiler which can convert the given C++ file into an executable ARM assembly program.

Any compiler follows certain process involving :

- Lexical Phase
- Syntax Parser
- Semantic Parser
- Intermediate Code Generation
- ICG Optimization
- Assembly Code Generation.

Our project involves performing all the above processes step by step in different phases.

Architecture of Language

This project deals with building a simple C++ compiler using Flex for the lexical phase, Yacc for abstract syntax tree and intermediate code generation and python for code optimization and assembly code generation.

The C++ compiler built deals with the following conditional and iterative programming constructs:

1. Conditional Constructs:
    a. if
    b. if-else
2. Iterative Constructs:
    a. for
    b. while

A C++ input file is given as an input to the compiler as follows:

Literature Survey

The following sources were referenced for this project:

https://www.javatpoint.com/lex

https://www.geeksforgeeks.org/introduction-to-yacc/

https://ruslanspivak.com/lsbasi-part7/

https://compileroptimizations.com/category/constant_propagation.htm

https://compileroptimizations.com/category/constant_folding.htm

https://compileroptimizations.com/category/dead_code_elimination.htm

https://www.javatpoint.com/code-generation

https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf

https://compileroptimizations.com/category/constant_propagation.htm

Context-Free Grammar

```
S : program
      ;

program
      : HASH INCLUDE '<' libraries '>'  program
      | HASH INCLUDE HEADER_LITERAL        program
      | NSPACE translation_unit
      ;
NSPACE
      : NAMESPACE ';'
translation_unit
      : ext_dec
      | translation_unit ext_dec
      ;


ext_dec
      : declaration
      | function_definition
      ;


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


compound_statement
      : '{' '}'
      | '{' block_item_list '}'
      ;


block_item_list
      : block_item
      | block_item_list block_item
      ;


block_item
      : declaration
      | statement
      | RETURN expression_statement
      | function_call ';'
      | printstat ';'
      ;
```

```
printstat
      : PRINT '(' STRING_LITERAL ')'
      | PRINT '(' STRING_LITERAL ',' expression ')'
      | COUT '<''<' STRING_LITERAL
      ;


declaration
      : type_specifier init_declarator_list ';'
      ;


statement
      : compound_statement
      | expression_statement
      | iteration_statement
      | conditional_statement
      ;

conditional_statement
      : IF '('
        conditional_expression ')'
        statement
        conditional_statement;
      | ELSE
      |
      ;

iteration_statement
      :FOR '(' expression_statement
      expression_statement
      expr  ')' statement

      | WHILE '(' conditional_expression ')'

      ;


expr
      : IDENTIFIER INC_OP
      | IDENTIFIER DEC_OP
      | INC_OP IDENTIFIER
      | DEC_OP IDENTIFIER
      | IDENTIFIER ADD_ASSIGN INTEGER_LITERAL
      | IDENTIFIER SUB_ASSIGN INTEGER_LITERAL
      | IDENTIFIER MUL_ASSIGN INTEGER_LITERAL
      | IDENTIFIER DIV_ASSIGN INTEGER_LITERAL
      | IDENTIFIER '=' IDENTIFIER '+' INTEGER_LITERAL
      | IDENTIFIER '=' IDENTIFIER '-' INTEGER_LITERAL
      | IDENTIFIER '=' IDENTIFIER '*' INTEGER_LITERAL
      | IDENTIFIER '=' IDENTIFIER '/' INTEGER_LITERAL
      ;
```

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


init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator
      ;


init_declarator
      : IDENTIFIER  '=' assignment_expression
      | IDENTIFIER
      ;

assignment_expression
      : conditional_expression
      | unary_expression assignment_operator assignment_expression

      ;

assignment_operator
      : '='
      | ADD_ASSIGN
      | SUB_ASSIGN
      | MUL_ASSIGN
      | DIV_ASSIGN
      | MOD_ASSIGN      ;
conditional_expression
      : equality_expression
      | equality_expression
      '?' expression
      ':' conditional_expression
      ;

expression_statement
      : ';'
      | expression ';'
      ;


expression
      : assignment_expression
      | expression ',' assignment_expression
      ;


primary_expression
      : IDENTIFIER
      | INTEGER_LITERAL
      | FLOAT_LITERAL
```

```
        | CHARACTER_LITERAL
        | '(' expression ')'
        ;

postfix_expression
        : primary_expression
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

unary_expression
        : postfix_expression
        | unary_operator unary_expression
        ;

unary_operator
        : '+'
        | '-'
        | '!'
        | '~'
        | "INC_OP"
        | "DEC_OP"
        ;
equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_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
        ;

additive_expression
        : multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression
        ;

multiplicative_expression
        : unary_expression
        | multiplicative_expression '*' unary_expression

        | multiplicative_expression '/' unary_expression

        | multiplicative_expression '%' unary_expression

        ;

function_definition
```

```
        : type_specifier declarator compound_statement

        | declarator compound_statement

        ;

function_call
        : declarator '(' identifier_list ')'
        | declarator '(' ')'
        ;

declarator
        : IDENTIFIER
        | declarator '(' parameter_list ')'
        | declarator '(' identifier_list ')'
        | declarator '(' ')'
        ;


parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
        ;


parameter_declaration
        : type_specifier IDENTIFIER
        | type_specifier
        ;


identifier_list
        : IDENTIFIER
        | identifier_list ',' IDENTIFIER
        ;
```

Design Strategies

- **Symbol Table Generation:**
  - In any compiler the symbol table holds details of the identifiers such as the scope,value,identifier name,line number etc. In our implementation of the symbol table,each entry of the symbol table is a structure whose properties include-token type,token value,line # and data type. Scope level is taken care of by a global variable initialized to 0 and is incremented every time the lex encounters an open curly bracket and decrements everytime the lex encounters a closed curly bracket.

- **Abstract Syntax Tree:**
  - The abstract syntax tree is constructed by first creating a node data structure containing necessary information of the token which will be stored in the AST. Along with the tokens generated by the lex the yacc is used for rule matching and nodes are inserted using the rule part of the grammar in yacc. Implicit data type conversion is done using the symbol table,everytime an assignment operation is encountered based on the value being assigned and the data type stored in the symbol table, the conversion is done.

- **Intermediate Code Generation:**
  - Intermediate code generation is done in formats in our implementation - 3 address code and quadruple format. For either methods the strategy mainly involves the yacc grammar. Our strategy was to generate the ICG component by component of any statement . For example for generating the ICG for a while loop, first the ICG is generated for the conditional expression inside the while loop parentheses and then generate the ICG individually for the statements inside the while loop.

- **Error Handling:**
  - As far as our implementation is concerned only basic error handling is taken care of during the syntax validation phase of our compiler. If a rule is not matched, the yacc implicitly makes a call to the yyerror() function where a char pointer is passed as an argument(argument is passed implicitly by the yacc). Based on the error and line number the error is printed.

- **Target Code Generation:**
  - Target code generated using a python script which reads the icg file and stores it line by line.Line by line the ARM statements are generated and registers are chosen in a round robin fashion(R%13) to ensure that registers are correctly used. For branch conditions, the condition statements are skipped and taken care off when we encounter the *ifFalse* statement. When the ifFalse statement is encountered then the previous statement which will be the conditional statement is converted into an ARM *CMP* statement and the ifFalse is converted to a *B* statement based on the condition.

Implementation Details

- **Symbol Table Creation:**
  - Tools used: Flex and GCC
  - Language used: C
  - Data Structures: Custom struct in C to hold symbol table entry details and array of this structure is used to hold symbol table
  - Commands to compile and run the lexical phase:

```
lex phase1.l
gcc lex.yy.c
./a.out < [filename]
```

- **Abstract Syntax Tree:**
  - Tools used: Flex,Bison Yacc and GCC
  - Language used: C
  - Data Structures: Internally stored as a binary tree
  - Commands to compile and run the semantic phase:

```
lex ast.l
yacc -d ast.y
gcc lex.yy.c y.tab.c
./a.out < [filename]
```

- **Intermediate Code Generation:**
  - Tools used: Flex,Bison Yacc and GCC
  - Language used: C
  - Data Structures: No special data structures are used implicitly.
    - Grammar rules are used to generate ICG component wise
    - Each statement of ICG is written to a text file
  - Commands to compile and run the ICG generator:

```
lex icg.l
yacc -d icg.y
gcc lex.yy.c y.tab.c
./a.out < [filename]
```

- **Code Optimization:**
  - Tools used: Python
  - Language used: Python

- Data Structures: Python list is used to hold list of lines
    - ICG file is read and split into sentences and stored in array
    - After processing,these statements are written back to a text file line by line
- Commands to run the python optimizer script:

```
python icg_opt.py [filename] --print
```

- **Assembly Code Generation:**
  - Tools used: Python
  - Language used: Python
  - Data structures:
    - Hash map is used to store a variable list to ensure variables are not loaded from memory again and again.
    - Python list to store ARM generated statements which are written to a **.s** ASM file after processing all ICG statements
  - Commands to run the assembly generator:

```
python gen.py [filename]
```

  Note: Assembly code will be stored in *filename.s*

- **Error Handling:**
  - Tools used: Yacc
  - Language used: C
  - Data structures: None
  - Command to run syntax analyzer:

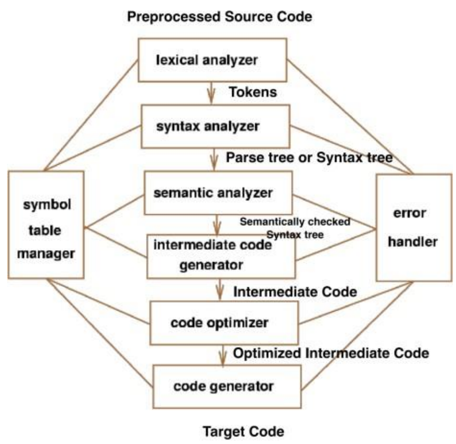
```
lex lexical.l
yacc -d gram.y
gcc lex.yy.c y.tab.c
./a.out < [filename]
```

Results

A simple C++ compiler with the following constructs was successfully built:

- ➔ for
- ➔ while
- ➔ if
- ➔ if-else

The implemented C++ compiler successfully performs syntax and semantic validation,ICG generation,ICG optimization and assembly code generation as per the below diagram:



An end to end (C++ to ARM) is shown below:

```
#include<iostream>
using namespace std;
int main()
{
    int i=0;
    int a=0;
    for(i=0;i<10;i++)
    {
        a=a+i;
    }
    a=2*a-1;
}
```

Input C++ File

```
.text
L0:
MOV R0,=i
MOV R1,[R0]
CMP R1,#10
BGE L1
MOV R2,=a
MOV R3,[R2]
MOV R4,=i
MOV R5,[R4]
MOV R6,=t1
MOV R7,[R6]
ADD R7,R3,R5
STR R7, [R6]
MOV R8,=i
MOV R9,[R8]
MOV R10,=t2
MOV R11,[R10]
ADD R11,#9,R1
STR R11, [R10]
MOV R12,=i
MOV R0,[R12]
MOV R1,#t2
STR R1, [R12]
goto L0
L1:
MOV R2,=a
MOV R3,[R2]
MOV R4,=t3
MOV R5,[R4]
MUL R5,#2,R3
STR R5, [R4]
MOV R6,=t3
MOV R7,[R6]
MOV R8,=t4
MOV R9,[R8]
SUBS R9,#7,R1
STR R9, [R8]
MOV R10,=a
MOV R11,[R10]
MOV R12,#t4
STR R12, [R10]
SWI 0x011
.DATA
i: .WORD 0
a: .WORD t1
```

ARM Assembly Generated for C++ file

Draw Drawback:

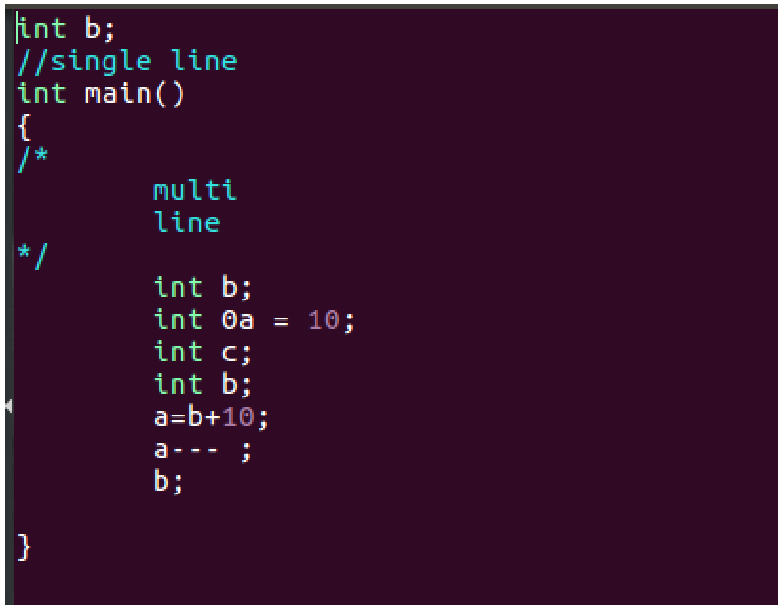
- Assembly code generated use too many registers as it reloads variables again and again

Future Work:

- Include more number of constructs
- Implement arrays
- Implement classes and objects

Snapshots

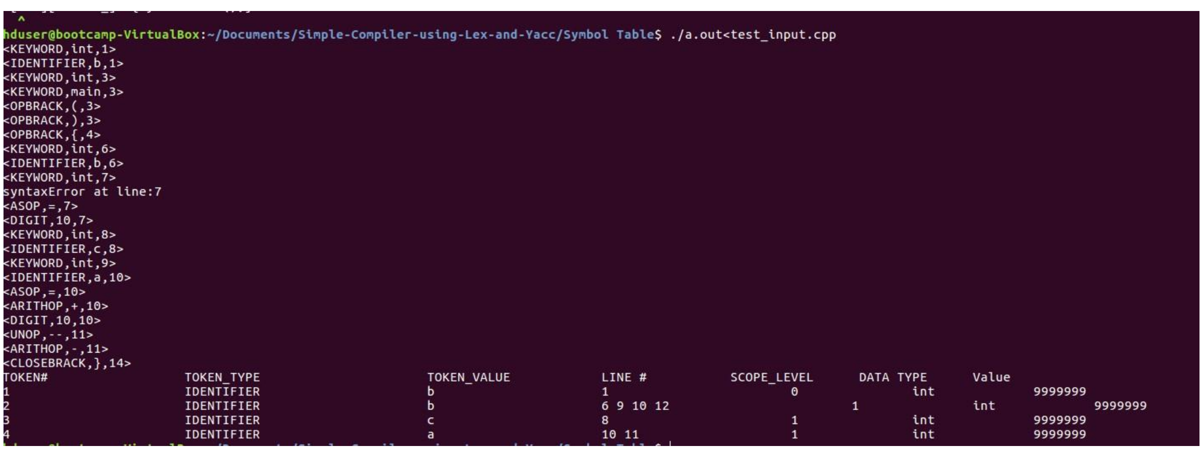
Phase 1: Lexical Phase(Generation of Tokens and Symbol Table)



```
int b;
//single line
int main()
{
/*
        multi
        line
*/
        int b;
        int 0a = 10;
        int c;
        int b;
        a=b+10;
        a--- ;
        b;

}
```

Input File



```
hduser@bootcamp-VirtualBox:~/Documents/Simple-Compiler-using-Lex-and-Yacc/Symbol Table$ ./a.out<test_input.cpp
<KEYWORD,int,1>
<IDENTIFIER,b,1>
<KEYWORD,int,3>
<KEYWORD,main,3>
<OPBRACK,(,3>
<OPBRACK,),3>
<OPBRACK,{,4>
<KEYWORD,int,6>
<IDENTIFIER,b,6>
<KEYWORD,int,7>
syntaxError at line:7
<ASOP,=,7>
<DIGIT,10,7>
<KEYWORD,int,8>
<IDENTIFIER,c,8>
<KEYWORD,int,9>
<IDENTIFIER,a,10>
<ASOP,=,10>
<ARITHOP,+,10>
<DIGIT,10,10>
<UNOP,--,11>
<ARITHOP,-,11>
<CLOSEBRACK,},14>
TOKEN#          TOKEN_TYPE          TOKEN_VALUE       LINE #        SCOPE_LEVEL      DATA TYPE      Value
1               IDENTIFIER          b                 1                     0              int              9999999
2               IDENTIFIER          b                 6 9 10 12                   1              int              9999999
3               IDENTIFIER          c                 8                     1              int              9999999
4               IDENTIFIER          a                 10 11                 1              int              9999999
```

Output result- Tokens generated and Symbol Table

Phase 2: Syntax Analyzer(Validation of Input With Grammar)

```cpp
int main(){
    //single line comment
    int nume=3;
    int a=10;
    int i=0;
    for(i=0;i<10;i++)
    {
        a=a+i;
    }
    /*
        multi line
    */
    int count=10;
    while(count1)
    {
        count--;
    }
    if(count==0)
    {
        count=count+2;
    }
    else
    {
        count=0;
    }

}
```

C++ code with error

```
user@DESKTOP-8PNH3AD:/mnt/c/Users/navne/Documents/Simple-Compiler-using-Lex-and-Yacc/Phase2$ ./a.out<t2.cpp
decl:int
main
decl:int
id:nume
assignop:=
num:3
decl:int
id:a
assignop:=
num:10
decl:int
id:i
assignop:=
num:0
for
id:i
assignop:=
num:0
id:i
compop:<
num:10
id:i
unary:++
id:a
assignop:=
id:a
id:i
decl:int
id:count
assignop:=
num:10
while
id:count1
Line no: 14
 The error is: syntax error, unexpected ')', expecting comparisionop
id:count
unary:--
if
id:count
compop:==
num:0
id:count
assignop:=
id:count
num:2
id:else
id:count
assignop:=
num:0
--------------

Symbol Table
Symbol          Scope           dtype           Value
nume            1               int             3
a               1               int             10
i               1               int             0
count           1               int             0
```

Syntax Error generated by parser along with tokens and symbol table

Phase 3: Semantic Analyzer(Abstract Syntax Tree Generation)

```cpp
int main(){

    int i, a, b;
    int nume=3.45;
    for(i = 0;  i < 10; i++){
        a=i;
    }
    i=1;
}
```

Input C++ Code

```
user@DESKTOP-8PNH3AD:/mnt/c/Users/navne/Documents/Simple-Compiler-using-Lex-and-Yacc/AST$ ./a.out<input.cpp

Line:4: 'float' to 'int'


ST

   SYMBOL              NAME      TYPE        SCOPE        LINE #            VALUE
 identifier               i       int          1             3                1
 identifier               a       int          1             3                0
 identifier               b       int          1             3                -
 identifier            nume       int          1             4                3



Abstract Syntax Tree

                                                                main

                                                        stmt

                                                stmt            =

                                        =           for        i          1

                                i          0          ++          =

                                        <          i          a          i

                             i          10

main     ( stmt     ( stmt     ( =   i   0   )     ( for     ( ++     ( <   i   10   )   i   )     ( =   a   i   )   )   )     ( =   i   1   )   )
```

Implicit float to int conversion
Symbol Table
AST along with the pre-order traversal

Phase 4: Intermediate Code Generation(3 address code and quadruple format)

```cpp
#include<iostream>
using namespace std;
int main()
{
    int i=2;
    if(i>1)
    {
        i=i+1;
    }
    else
    {
        i=i-1;
    }
    i=i+3;
    int a;
    for(i=0;i<10;i++)
    {
        a=i+2;
    }
    a=a*3+4;
    i=1;
    while(i<11)
    {
        a=i-2;
    }
    a=i+2*a;
    cout<<"End";
}
```

Input C++ File

```
Intermediate Code

i = 2
t0 = i > 1
ifFalse t0 goto L0
t1 = i + 1
i = t1
goto L1
L0:
t2 = i - 1
i = t2
L1:
t3 = i + 3
i = t3
i = t3
L2:
t4 = i < 10
ifFalse t4 goto L3
t5 = i + 2
a = t5
t6 = i + 1
i = t6
goto L2
L3:
t7 = a * 3
t8 = t7 + 4
a = t8
i = t8
L4:
t9 = i < 11
ifFalse t9 goto L5
t10 = i - 2
a = t10
goto L4
L5:
t11 = 2 * a
t12 = i + t11
a = t12
```

3 address code ICG

Quadruple Format

Op	Arg1	Arg2	Res
=	2		i
>	i	1	t0
ifFalse	t0		L0
+	i	1	t1
=	t1		i
goto			L0
Label			L0
-	i	1	t2
=	t2		i
Label			L1
+	i	3	t3
=	t3		i
=	t3		i
Label			L2
<	i	10	t4
ifFalse	t4		L3
+	i	2	t5
=	t5		a
+	i	1	t6
=	t6		i
goto			L2
Label			L3
*	a	3	t7
+	t7	4	t8
=	t8		a
=	t8		i
Label			L4
<	i	11	t9
ifFalse	t9		L5
-	i	2	t10
=	t10		a
goto			L4
Label			L5
*	2	a	t11
+	i	t11	t12
=	t12		a

Quadruple Format ICG

Phase 5: ICG Optimization(Constant Propagation,Constant folding and Dead Code Elimination)

```
t0 = 1
t1 = 3
t3 = t0 + t1
t1 = 2
a = t3 + t4
```

Example ICG Before Optimization

```
After Constant Propagation
t0 = 1
t1 = 3
t3 = 1 + 3
t1 = 2
a = t3 + t4


After Constant Folding:
t0 = 1
t1 = 3
t3 = 4
t1 = 2
a = t3 + t4


After Dead Code Elimination:
t3 = 4
a = t3 + t4
```

Output of constant propagation,constant folding and dead code elimination

Phase 6: Assembly Code Generation(ARM Instruction Set)

```
t0 = 1
t1 = 2
t2 = t1 + t2
t3 = t1 < t2
ifFalse t3 goto L0:
t3 = t3 + 1
L0:
t3 = t3 - 1
```

ICG code to be converted to target code

```
.text
MOV R0,=t1
MOV R1,[R0]
MOV R2,=t2
MOV R3,[R2]
MOV R4,=t2
MOV R5,[R4]
ADD R5,R1,R3
STR R5, [R4]
MOV R6,=t1
MOV R7,[R6]
MOV R8,=t2
MOV R9,[R8]
CMP R7,R9
BGE L0:
MOV R10,=t3
MOV R11,[R10]
MOV R12,=t3
MOV R0,[R12]
ADD R0,#11,R1
STR R0, [R12]
L0:
MOV R1,=t3
MOV R2,[R1]
MOV R3,=t3
MOV R4,[R3]
SUBS R4,#2,R1
STR R4, [R3]
SWI 0x011
.DATA
t0: .WORD 1
t1: .WORD 2
```

ARM assembly code generated

Conclusion:

A simple C++ compiler was successfully built for the previously mentioned constructs as well as all basic C++ programming paradigms.