VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY BURLA

DEPARTMENT OF INFORMATION TECHNOLOGY

PROJECT REPORT ON

ALPHA C Compiler

Submitted by:

SOUMYADEEP PANI	2102081042
ANWESH CHHATOI	2102081063
ROHIT PATANAIK	2102081034
ABHINEET SETH	2102081048
SHANKAR KISAN	2102081058

Under Esteemed Guidance Of

Dr. D.Chandrasekhar Rao
(Assistant Professor)
&
Ms.Chinmayee Mohapatra

DEPARTMENT OF INFORMATION TECHNOLOGY

VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY BURLA

CERTIFICATE

This is to certify that that the project entitled "ALPHA" a compiler for C programming Language is submitted by Mr. Soumyadeep Pani (2102081042), Mr. Anwesh Chhatoi (2102081063), Mr. Rohit Patanaik (2102081034), Mr. Abhineet Seth (2102081048) and Mr. Shankar Kishan (2102081058) from the Department of Information Technology, Veer Surendra Sai University of Technology, Burla, in partial fulfilment of the requirements for the degree of Bachelor of Technology. It is an original report carried out by them under our supervision and guidance.

Dr. P.K. Sahu	Dr. D.Chandrasekhar Rao
(Head of Information Technology)	(Assistant professor)
VSSUT, BURLA	

ACKNOWLEDGEMENT

We would like to thank our supervisor Dr. D. Chandrasekhar Rao for giving us the opportunity to undertake this project. We are extremely grateful for his insightful guidance, timely advice, continuous support and motivation throughout our candidature. We especially thank him for his kindness and excellent contributions

We would like to thank all the faculty and staff members of the Department of Information Technology at Veer Surendra Sai University of Technology, Burla, India, for their help and support. We are deeply thankful to Prof. Bansidhar Majhi, Vice-Chancellor, and Dr. P.K. Sahu, Head of the Department, Department of Information Technology, for their moral support and providing us with all the necessary facilities during our candidature. We would like to thank our family members for their unconditional love and support.

Soumyadeep Pani

(2102081042)

Anwesh Chhatoi

(2102081063)

Rohit Patanaik

(2102081034)

Abhineet Seth

(2102081048)

Shankar Kisan

(2102081058)

TABLE OF CONTENTS

OBJECTIVE

The objective of this Compiler Design Lab project report is to document the comprehensive process involved in designing and implementing a compiler for the C programming language. The primary aim is to provide a detailed account of the steps taken to construct a functional compiler that translates C source code into executable machine code. This report will elucidate the theoretical concepts, algorithmic strategies, and practical techniques employed in each phase of the compiler construction process, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Furthermore, the report will analyze the challenges encountered during the development process and evaluate the performance and efficiency of the implemented compiler through rigorous testing and benchmarking against standard C compilers. Ultimately, this project aims to deepen understanding of compiler design principles and methodologies while demonstrating proficiency in applying theoretical knowledge to practical implementation tasks.

INTRODUCTION :-

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

This is done using the following steps:

- Generate symbol table after performing expression evaluation
- Generate Abstract Syntax Tree for the code
- Generate 3 address code followed by corresponding quadruples
- Perform Code Optimization

The main tools used in the project include LEX which identifies predefined patterns and generates tokens for the patterns matched and YACC which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code for the source code.
PYTHON is used to optimize the intermediate code generated by the parser.

ARCHITECTURE OF LANGUAGE

C++ constructs implemented:

1. Simple If
2. If-else
3. Ternary operator
4. While loop
5. For-loop

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

DESIGN STAGES AND IMPLEMENTATION

Phase 1: (a)Lexical Analysis

- LEX tool was used to create a scanner for C++ language
- The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
- The scanner also scans for the comments (single-line and multiline comments) and writes the source file without comments onto an output file which is used in the further stages.
- All tokens included are of the form T_<token-name>.Eg: T_pl for '+',T_min for '-', T_lt for '<' etc.
- A global variable 'yylavl' is used to record the value of each lexeme scanned. 'yytext' is the lex variable that stores the matched string.
- Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- The rules are regular expressions which have corresponding actions that execute on a match with the source input.

The following is the lex file used -

```
%{
    #include<string.h>
    #include<stdio.h>
int line = 0;     #define
YYSTYPE char *
%}  alpha [A-Za-
z_] digit [0-9]
%option yylineno
%%
[ \t\n] {yylval = strdup(yytext);}
":" {yylval = strdup(yytext);return T_colon;}
"?" {yylval = strdup(yytext);return T_ques;}
"while" {yylval = strdup(yytext);return WHILE;}
"for" {yylval = strdup(yytext);return FOR;}
"if" {yylval = strdup(yytext);return IF;}
"else" {yylval = strdup(yytext);return ELSE;}
"cout" {yylval = strdup(yytext);return COUT;}
"endl" {yylval = strdup(yytext);return ENDL;}
"break" {yylval = strdup(yytext);return BREAK;}
"continue" {yylval = strdup(yytext);return CONTINUE;}
"int" {yylval = strdup(yytext);return INT;}
"float" {yylval = strdup(yytext);return FLOAT;}
"char" {yylval = strdup(yytext);return CHAR;}
"void" {yylval = strdup(yytext);return VOID;}
"#include" {yylval = strdup(yytext);return INCLUDE;}
"main()" {yylval = strdup(yytext);return MAINTOK;}
{digit}+    {yylval = strdup(yytext);return NUM;}
{digit}+.{digit}+ {yylval = strdup(yytext);return FLOAT;}
{alpha}({alpha}|{digit})* {yylval = strdup(yytext);return ID;}
{alpha}({alpha}|{digit})*"\.h"?  {yylval = strdup(yytext);return H;}
\".*\" {yylval = strdup(yytext);return STRING;}
"<" {yylval = strdup(yytext);return T_lt;}
">"     {yylval = strdup(yytext);return T_gt;}
"="     {yylval = strdup(yytext);return T_eq;}
"<="    {yylval = strdup(yytext);return T_lteq;}
">="    {yylval = strdup(yytext);return T_gteq;}
"=="    {yylval = strdup(yytext);return T_eqeq;}
"!="    {yylval = strdup(yytext);return T_neq;}
"+"     {yylval = strdup(yytext);return T_pl;} "-"
{yylval = strdup(yytext);return T_min;}
"*"     {yylval = strdup(yytext);return T_mul;}
"/"     {yylval = strdup(yytext);return T_div;}
"++"    {yylval = strdup(yytext);return T_incr;}
"--"    {yylval = strdup(yytext);return T_decr;}
"!"     {yylval = strdup(yytext);return T_neq;}
"||"    {yylval = strdup(yytext);return T_or;}
"&&"    {yylval = strdup(yytext);return T_and;}
.    return yytext[0];

%%
```

Phase 1: (b)Syntax Analysis

- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- The design implementation supports
    1. Variable declarations and initializations
    2. Variables of type int,float and char
    3. Arithmetic and boolean expressions
    4. Postfix and prefix expressions
    5. Constructs - **if-else, ternary, while loop and for loop**
- Yacc tool os used for parsing. It reports shift-reduce and reducereduce conflicts on parsing an ambiguous grammar.

The following is the CFG used -

```
S
      : START
      ;

START
      : INCLUDE T_lt H T_gt MAIN
      | INCLUDE "\"" H "\"" MAIN
      ;

MAIN
```

```
        : VOID MAINTOK BODY
        | INT MAINTOK BODY
        ;

BODY
        : '{' C '}'
        ;

C
        : C statement ';'
        | C LOOPS
        | statement ';'
        | LOOPS
        ;

LOOPS
        : WHILE '(' COND ')' LOOPBODY
        | FOR '(' ASSIGN_EXPR ';' COND ';' statement ')' LOOPBODY
        | IF '(' COND ')' LOOPBODY
        | IF '(' COND ')' LOOPBODY ELSE LOOPBODY
        ;


LOOPBODY
        : '{' LOOPC '}'
        | ';'
        | statement ';'
        ;

LOOPC
        : LOOPC statement ';'
        | LOOPC LOOPS
        | statement ';'
        | LOOPS
        ;  statement
: ASSIGN_EXPR
        | EXP
        | TERNARY_EXPR
        | PRINT
        ;

COND
        : LIT RELOP LIT
        | LIT
        | LIT RELOP LIT bin_boolop LIT RELOP LIT
        | un_boolop '(' LIT RELOP LIT ')'
```

```
      | un_boolop LIT RELOP LIT
      | LIT bin_boolop LIT
      | un_boolop '(' LIT ')'
      | un_boolop LIT
      ;

ASSIGN_EXPR
      : ID T_eq EXP
      | TYPE ID T_eq EXP
      ;

EXP
      : ADDSUB
      | EXP T_lt ADDSUB
      | EXP T_gt ADDSUB
      ;

ADDSUB
      : TERM
      | EXP T_pl TERM
      | EXP T_min TERM
      ;

TERM
      : FACTOR
      | TERM T_mul FACTOR
      | TERM T_div FACTOR
      ;

FACTOR
      : LIT
      | '(' EXP ')'
      ;

TERNARY_EXPR
      : '(' COND ')' T_ques ternary_statement
      ;
ternary_statement
      : statement T_colon statement
      ;

PRINT
      : COUT T_lt T_lt STRING
      | COUT T_lt T_lt STRING T_lt T_lt ENDL
      ;
LIT
```

```
        : ID
        | NUM
        ;
TYPE
        : INT
        | CHAR
        | FLOAT
        ;
RELOP
        : T_lt
        | T_gt
        | T_lteq
        | T_gteq
        | T_neq
        | T_eqeq
        ;
bin_boolop
: T_and
        | T_or
        ;
un_arop
: T_incr
        | T_decr
        ;
un_boolop
: T_not
        ;
```

Phase 2: Symbol table with expression evaluation :-

- A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurence, the category of the token (constant, variable,keyword,operator),the value that it holds the datatype.

```
typedef struct symbol_table
  {     int line;
char name[31];
char type;
char *value;
char *datatype;
  }
ST;
```

- As each line is parsed, the actions associated with the grammar rules is executed. Symbol tables functions such as lookup, search_id, update and get_val are called appropriately with each production rule.
- $1 is used to refer to the first token in the given production and $$ is used to refer to the resultant of the given production.
- Expressions are evaluated and the values of the used variables are updated accordingly.
- At the end of the parsing, the updated symbol table is displayed.

For the following input, the corresponding symbol table generated is shown:

```c
1 #include<stdio.h>
2 void main()
3 {
4         int a = 4 * 5 / 2;
5         int b = a * 7;
6
7         int c = a / b + 8 / 4;
8         int d = a + b * c;
9         b = 100 * 100 - d + c;
10
11 }
12
```

```
INPUT ACCEPTED.
Parsing Complete
Number of entries in the symbol table = 19

----------------------------------Symbol Table----------------------------------
```

S.No	Token	Line Number	Category	DataType	Value
1	int	8	keyword	NULL	(null)
2	4	7	constant	NULL	(null)
3	5	4	constant	NULL	(null)
4	*	9	operator	NULL	(null)
5	2	4	constant	NULL	(null)
6	/	7	operator	NULL	(null)
7	a	4	identifier	int	10
8	=	9	operator	NULL	(null)
9	7	5	constant	NULL	(null)
10	b	9	identifier	int	9852
11	8	7	constant	NULL	(null)
12	+	9	operator	NULL	(null)
13	c	7	identifier	int	2
14	d	8	identifier	int	150
15	100	9	constant	NULL	(null)
16	-	9	operator	NULL	(null)
17	void	2	keyword	NULL	(null)
18	main()	2	keyword	NULL	(null)
19	#include	1	keyword	NULL	(null)

Phase 3: Abstract Syntax Tree:-

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

```
%left T_lt T_gt
%left T_pl T_min
%left T_mul T_div
```

To build the tree, a structure is maintained which has pointers to its children and a container for its data value.

```
 typedef struct
Abstract_syntax_tree
  {     char
*name;
        struct Abstract_syntax_tree *left;
  struct Abstract_syntax_tree *right;
  }node;
```

When every new token is encountered during parsing, the buildTree function takes in the value of the token, creates a node of the tree and attaches it to its parent(head of the reduced production). When the head production of the construct is reached the printTree function displays the tree for it. A node named SEQ is used to connect consecutive statements in the construct that are not related.

Sample Input 1:

```
 if ( a < b
)
        {
a = 10;
b = 2 * 3;
                a = 0;
        }
```

Sample Output 1:

```
( IF ( <  a  b )( SEQ ( SEQ ( =  a  10 )( =  b ( *  2  3 )))( =  a  0 )))
```

Sample Input 2:

```
#include<stdio.h>
void main()
{
        int a = 4 * 5 / 2;
        int b = a * 7;

        while( a>b ){
                a = a+1;
        }

        int x = 20*a;

        if( b <= x ){
                a = 10;
        }

        a = 100;
        int i = 1;


    int y = a+b;

    (x < b) ? x = 10 : x=11;

}
```

Sample Output 2:

```
( =  a ( / ( *  4  5 ) 2 ))
----------------------------------------------------------------
( =  b ( *  a  7 ))
----------------------------------------------------------------
( WHILE ( >  a  b )( =  a ( +  a  1 )))
----------------------------------------------------------------
( =  x ( *  20  a ))
----------------------------------------------------------------
( IF ( <=  b  x )( =  a  10 ))
----------------------------------------------------------------
( =  a  100 )
----------------------------------------------------------------
( =  i  1 )
----------------------------------------------------------------
( =  y ( +  a  b ))
----------------------------------------------------------------
( ? ( <  x  b )( : ( =  x  10 )( =  x  11 )))
----------------------------------------------------------------
Input accepted.
Parsing Complete
```

Phase 4: Intermediate Code Generation (ICG)

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.

Three-Address Code –
A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have an address (memory location).

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

```
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
```

T1, T2, T3 are temporary variables.

The data structure used to represent Three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

Sample Input:

```
#include<stdio.h>
void main()
{
        int b = a*b;

        while( a>b ){
                a = a+1;
        }

        x = 20*a;

        if( b < = c ){
                a = 10;
        }
        else{
                a = 20;
        }

        a = 100;

    for(i=0;i<10;i = i+1){
            a = a+1;
    }

    y = a+b;

    (x < b) ? x = 10 : x=11;

}
```

Sample Output:

1. Three Address Code

```
sandya@ubuntu: ~/Desktop/CD lab/phase4/ICG
                                        ^^^^^^^^
T0 = 4 * 5
T1 = T0 / 2
a = T1
T2 = a * 7
b = T2
L0:
T3 = a > b
T4 = not T3
if T4 goto L1
T5 = a + 1
a = T5
goto L0
L1:
T6 = 20 * a
x = T6
T7 = b <= x
T8 = not T7
if T8 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 1
i = 0
L5:
T9 = i < 10
T10 = not T9
if T10 goto L6
goto L7
L8:
T11 = i + 1
i = T11
goto L5
L7:
T12 = a + 1
a = T12
goto L8
L6:
T13 = a + b
y = T13
T14 = < < b
T15 = not T14
if T15 goto L9
x = 10
goto L10
L9:
x = 11
L10:
Input accepted.
Parsing Complete
```

2. Quadruples

```
sandya@ubuntu: ~/Desktop/CD lab/phase4/ICG
Operator        Arg1            Arg2            Result
*               4               5               T0
/               T0              2               T1
=               T1              (null)          a
*               a               7               T2
=               T2              (null)          b
Label           (null)          (null)          L0
>               a               b               T3
not             T3              (null)          T4
if              T4              (null)          L1
+               a               1               T5
=               T5              (null)          a
goto            (null)          (null)          L0
Label           (null)          (null)          L1
*               20              a               T6
=               T6              (null)          x
<=              b               x               T7
not             T7              (null)          T8
if              T8              (null)          L3
=               10              (null)          a
goto            (null)          (null)          L4
Label           (null)          (null)          L3
=               20              (null)          a
Label           (null)          (null)          L4
=               100             (null)          a
=               1               (null)          i
=               0               (null)          i
Label           (null)          (null)          L5
<               i               10              T9
not             T9              (null)          T10
if              T10             (null)          L6
goto            (null)          (null)          L7
Label           (null)          (null)          L8
+               i               1               T11
=               T11             (null)          i
goto            (null)          (null)          L5
Label           (null)          (null)          L7
+               a               1               T12
=               T12             (null)          a
goto            (null)          (null)          L8
Label           (null)          (null)          L6
+               a               b               T13
=               T13             (null)          y
<               <               b               T14
not             T14             (null)          T15
if              T15             (null)          L9
=               10              (null)          x
goto            (null)          (null)          L10
Label           (null)          (null)          L9
=               11              (null)          x
Label           (null)          (null)          L10
sandya@ubuntu:~/Desktop/CD lab/phase4/ICG$
```

Phase 5: Code Optimization

The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values (possibly after expression evaluation). This structure is used to perform constant propagation and constant folding in sequential blocks followed by dead code elimination.

Sample Input(Quadruples)

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

Sample Input(3 Address Code)

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

Sample Output:

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:/Users/sandy/Desktop/codeopt.py =================
Quadruple form after Constant Folding
-------------------------------------
= 3 NULL a
= 8 NULL b
= 11 NULL c
* 11 e d
= 8 NULL a
= 16 NULL f
if x NULL L0


Constant folded expression -
--------------------
a = 3
b = 8
c = 11
d = 11 * e
a = 8
f = 16
if x goto L0


After dead code elimination -
------------------------------
d = 11 * e
if x goto L0
>>>
```

```
1 #include<stdio.h>
2 void main()
3 {
4         int a = 4 * 5 / 2;
5         int b = a * 7;
6 
7         while( a>b ){
8                 a = a+1;
9         }
10 
11        int x = 20*a;
12 
13        if( b <= x ){
14                a = 10;
15        }
16        else{
17                a = 20;
18        }
19        c = 40;
20        a = 100;
21        int i = 1;
22    for(i=0;i<10;i = i+1){
23            a = a+1;
24    }
25 
26    int y = a+b;
27 
28    (x < b) ? x = 10 : x=11;
```

```
sandya@ubuntu:~/Desktop/CD lab/symTable+exp$ sh phase2.sh
jughead.y: warning: 1 nonterminal useless in grammar [-Wother]
jughead.y: warning: 2 rules useless in grammar [-Wother]
jughead.y:163.1-7: warning: nonterminal useless in grammar: un_aro
 un_arop
 ^^^^^^^
jughead.y:164.9-55: warning: rule useless in grammar [-Wother]
       : T_incr {lookup($1,@1.last_line,'O',NULL,NULL);}
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
jughead.y:165.9-55: warning: rule useless in grammar [-Wother]
       | T_decr {lookup($1,@1.last_line,'O',NULL,NULL);}
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
jughead.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
jughead.y: warning: 9 reduce/reduce conflicts [-Wconflicts-rr]
Error at line 19 : c is not defined
sandya@ubuntu:~/Desktop/CD lab/symTable+exp$
```

This shows the detection of invalid syntax at line 21

```
1 #include<stdio.h>
2 void main()
3 {
4         int a = 4 * 5 / 2;
5         int b = a * 7;
6 
7         while( a>b ){
8                 a = a+1;
9         }
10 
11        int x = 20*a;
12 
13        if( b <= x ){
14                a = 10;
15        }
16        else{
17                a = 20;
18        }
19 
20 
21        a = 100;;
22        int i = 1;
23    for(i=0;i<10;i = i+1){
24            a = a+1;
25    }
26 
27    int y = a+b;
28 
29    (x < b) ? x = 10 : x=11;
30 
31 }
```

```
sandya@ubuntu:~/Desktop/CD lab/symTable+exp$ sh phase2.sh
jughead.y: warning: 1 nonterminal useless in grammar [-Wother]
jughead.y: warning: 2 rules useless in grammar [-Wother]
jughead.y:163.1-7: warning: nonterminal useless in grammar: un_arop [-Wother]
 un_arop
 ^^^^^^^
jughead.y:164.9-55: warning: rule useless in grammar [-Wother]
       : T_incr {lookup($1,@1.last_line,'O',NULL,NULL);}
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
jughead.y:165.9-55: warning: rule useless in grammar [-Wother]
       | T_decr {lookup($1,@1.last_line,'O',NULL,NULL);}
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
jughead.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
jughead.y: warning: 9 reduce/reduce conflicts [-Wconflicts-rr]
Error :; at 21
Parsing failed
sandya@ubuntu:~/Desktop/CD lab/symTable+exp$
```

CONCLUSION:-

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an intermediate code, given a C++ code as input.

There are a few shortcomings with respect to our implementation. The symbol table structure is same across all types of tokens (constants, identifiers and operators). This leads to some fields being empty for some of the tokens. This can be optimized by using a better representation.

The Code optimizer does not work well when propagating constants across branches (At if statements and loops). It works well only in sequential programs. This needs to be rectified.

FUTURE ENHANCEMENTS:-

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

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