PROJECT REPORT

CS C++ Mini Compiler

CSB 353: Compiler Design

***Submitted To*:**

Dr. Shelly Sachdeva
Associate Professor, CSE Department
NIT DELHI

Submitted by:

Deepak Sharma (201210014)
Anavi Somani (201210005)
Atul Goyal (201210011)

NATIONAL INSTITUTE OF TECHNOLOGY DELHI
Department of Computer Science and Engineering
2023

ACKNOWLEDGMENT

Completing this report would not have been possible without the support and assistance of numerous individuals and organizations, who deserve heartfelt appreciation and recognition. We would like to express our gratitude to each and every one of them.

First and foremost, we extend our deepest gratitude to Dr. Shelly Sachdeva, our supervisor, for her invaluable guidance, mentorship, and support throughout this project. Her expertise and insights have been instrumental in shaping our research and analysis, and we have learned immensely from her.

We appreciate MS. Kanika Soni's assistance and cooperation in helping to make this report thorough and useful. We are also grateful for your timely response to my queries, feedback, and suggestions.

We value your professionalism and meticulousness throughout the whole process, and we are certain that your contributions have improved the report's quality.

Your expertise in the field has been a great asset, and we feel privileged to have had the opportunity to work with you.

We also extend our sincere thanks to the staff and management of NIT Delhi , who generously provided us with access to their resources, facilities, and data. Their cooperation and assistance have been integral to our research, and we are grateful for their support.

Once again, thank you for your outstanding contribution, and we look forward to collaborating with you on future projects.

TABLE OF CONTENTS

CS C++ Mini Compiler

PROJECT REPORT

TITLE

The intended goal of this project is to create a c++ compiler that could interpret conditional statements like If-Else statements and Switch-Case constructs and output the results at each stage of compilation before producing 8086 assembly code as a final output.

OBJECTIVE

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 (for and while).
The main functionality of the project is to generate an optimized intermediate code for the given C++ source code and also assembly code using this optimized intermediate code generated.

This is done using the following steps:

i) Generate symbol table after performing expression evaluation
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code followed by corresponding quadruples
iv) Perform Code Optimization
v) Generate Assembly code

INTRODUCTION

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or 'code' that a computer's processor uses.

The intended goal of this project is to create a c++ compiler that could interpret conditional statements like If-Else statements and Switch-Case constructs and output the results at each stage of compilation before producing 8086 assembly code as a final output.

ARCHITECTURE OF LANGUAGE

C++ constructs implemented:

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

I. Arithmetic expressions with +, -, *, /, ++, -- are handled
II. Boolean expressions with >,<,>=,<=,== are handled
III. Error handling reports undeclared variables
IV. Error handling also reports syntax errors with line numbers
V. Error handling also reports if the same variable is declared twice in the same scope.

REQUIREMENTS

- Flex
- Bison / YACC
- Text Editor (VS Code)
- GCC Compiler
- Python

What does the compiler actually do?

It converts the high-level language to low level language or assembly code. Here for the compiler, we built takes the input of C++ code in .cpp. file format and give the results for each phase of the compiler. We have implemented all the phases of the compiler in our project. The final result of our compiler will be Assembly Code for the given input.

How to execute Program?

- Lex File

```
lex lexer.l
./a.out < input.cpp
```

- YACC File

```
lex scanner.l
yacc -d parser.y
gcc lex.yy.c y.tab.c
./a.out < input.cpp
```

- Python File

```
python3 icg_opt.py input.cpp
```

Team Members

Deepak Sharma: 201210014@nitdelhi.ac.in
Anavi Somani: 201210005@nitdelhi.ac.in
Atul Goyal: 201210011@nitdelhi.ac.in

DESIGN STAGES AND IMPLEMENTATION

Phase 1: Lexical Analysis

The lexer or scanner will transform 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 should consume any comments from the input stream and ignore them. If a file ends with an unterminated comment, the scanner should report an error.

Recording the position of each lexeme requires us to track the current line and column numbers (we will need global variables) and update them as the scanner reads the file, most likely incrementing the line count on each newline and the column on each token.

Our goal at this stage:

- skip over white space
- recognize all keywords and return the correct token
- recognize operators and return the correct token
- record the line number and first and last column in yylloc for all tokens
- report lexical errors for improper strings, lengthy identifiers, and invalid characters
- recognize identifiers, return the correct token and set appropriate fields of yylval.

We will use flex/lex to create a scanner for our programming language.

Scanner/Lexer implementation: The yylval global variable is used to record the value for each lexeme scanned and the yylloc global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code.

For each character that cannot be matched to any token pattern, it reports and continues parsing with the next character.

If a string erroneously contains a newline, report an error and continue at the beginning of the next line.

All tokens included are of the form T_<token-name>.Eg: T_pl for „+", T_min for „-", T_lt for „<" etc.

The rules are regular expressions which have corresponding actions that execute on a match with the source input.

LEX File:-

```
%{
int lineno = 1;
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
%}
alpha [A-Za-z]
digit [0-9]
und [_]
space [ ]
tab [	]
line [\n]
char \'.\'
at [@]
string \"(.^([%d]|[%f]|[%s]|[%c]))\"
%%
{space}* {}
{tab}* {}
{string} return STRING_CONST;
{char} return CHAR_CONST;
{line} {lineno++;}
auto return AUTO;
break return BREAK;
case return CASE;
char return CHAR;
const return CONST;
```

```
continue return CONTINUE;
default return DEFAULT;
do return DO;
double return DOUBLE;
else return ELSE;
enum return ENUM;
extern return EXTERN;
float return FLOAT;
for return FOR;
goto return GOTO;
if return IF;
int return INT;
long return LONG;
register return REGISTER;
return return RETURN;
short return SHORT;
signed return SIGNED;
sizeof return SIZEOF;
static return STATIC;
struct return STRUCT;
switch return SWITCH;
typedef return TYPEDEF;
union return UNION;
unsigned return UNSIGNED;
void return VOID;
volatile return VOLATILE;
while return WHILE;
```

```
printf return PRINTF;

scanf return SCANF;

using return USING;

namespace return NAMESPACE;

std return STD;

endl return ENDL;

cout return COUT;

cin return CIN;


{alpha}({alpha}|{digit}|{und})* return IDENTIFIER;

 [+-][0-9]{digit}*(\.{digit}+)? return SIGNED_CONST;


"//" return SLC;

"/*" return MLCS;

"*/" return MLCE;

"<=" return LEQ;

">=" return GEQ;

"==" return EQEQ;

"!=" return NEQ;

"||" return LOR;

"&&" return LAND;

"=" return ASSIGN;

"+" return PLUS;

"-" return SUB;

"*" return MULT;

"/" return DIV;

"%" return MOD;
```

```
"<" return LESSER;
">" return GREATER;
"++" return INCR;
"--" return DECR;
">>" return RTSFT;
"<<" return LFSFT;
"," return COMMA;
";" return SEMI;

"#include<iostream>" return HEADER;
"#include <stdio.h>" return HEADER;
"main()" return MAIN;

{digit}+ return INT_CONST;
({digit}+)\.({digit}+) return FLOAT_CONST;

"%d"|"%f"|"%u"|"%s" return TYPE_SPEC;
"\"" return DQ;
"(" return OBO;
")" return OBC;
"{" return CBO;
"}" return CBC;
"#" return HASH;

{alpha}({alpha}|{digit}|{und})*\[{digit}*\] return ARR;
{alpha}({alpha}|{digit}|{und})*\(({alpha}|{digit}|{und}|{space})*\) return FUNC;
({digit}+)\.({digit}+)\.({digit}|\.)* return NUM_ERR;
```

```
({digit}|{at})+({alpha}|{digit}|{und}|{at})* return UNKNOWN;

%%
```

OUTPUT:

```
● atul0607@atulg67:~/compiler/Phase1$ lex lexer.l
● atul0607@atulg67:~/compiler/Phase1$ ./a.out < input.cpp


 #include<iostream>       HEADER                         Line 1
 using                    KEYWORD                        Line 2
 namespace                        KEYWORD                        Line 2
 std                      KEYWORD                        Line 2
 ;                        SPECIAL SYMBOL                 Line 2
 int                      KEYWORD                        Line 4
 main()                   MAIN FUNCTION                  Line 4
 {                        SPECIAL SYMBOL                 Line 5
 int                      KEYWORD                        Line 6
 a                        IDENTIFIER                     Line 6
 ;                        SPECIAL SYMBOL                 Line 6
 cin                      PRE DEFINED FUNCTION           Line 7
 >>                       OPERATOR                       Line 7
 a                        IDENTIFIER                     Line 7
 ;                        SPECIAL SYMBOL                 Line 7
 while                    KEYWORD                        Line 8
 (                        SPECIAL SYMBOL                 Line 8
 a                        IDENTIFIER                     Line 8
 <                        OPERATOR                       Line 8
 10                       INTEGER CONSTANT               Line 8
 )                        SPECIAL SYMBOL                 Line 8
 {                        SPECIAL SYMBOL                 Line 9
 if                       KEYWORD                        Line 10
 (                        SPECIAL SYMBOL                 Line 10
 a                        IDENTIFIER                     Line 10
 %                        OPERATOR                       Line 10
 2                        INTEGER CONSTANT               Line 10
 ==                       OPERATOR                       Line 10
 0                        INTEGER CONSTANT               Line 10
 )                        SPECIAL SYMBOL                 Line 10
 {                        SPECIAL SYMBOL                 Line 11
 cout                     PRE DEFINED FUNCTION           Line 12
 <<                       OPERATOR                       Line 12
 "                        SPECIAL SYMBOL                 Line 12
 Even                     IDENTIFIER                     Line 12
 Number                   IDENTIFIER                     Line 12
 :"                       SPECIAL SYMBOL                 Line 12
 <<                       OPERATOR                       Line 12
 a                        IDENTIFIER                     Line 12
 <<                       OPERATOR                       Line 12
 endl                     KEYWORD                        Line 12
 ;                        SPECIAL SYMBOL                 Line 12
 }                        SPECIAL SYMBOL                 Line 13
 else                     KEYWORD                        Line 14
 {                        SPECIAL SYMBOL                 Line 15
 cout                     PRE DEFINED FUNCTION           Line 16
 <<                       OPERATOR                       Line 16
 "                        SPECIAL SYMBOL                 Line 16
 Odd                      IDENTIFIER                     Line 16
 Number                   IDENTIFIER                     Line 16
 :"                       SPECIAL SYMBOL                 Line 16
 <<                       OPERATOR                       Line 16
 a                        IDENTIFIER                     Line 16
 <<                       OPERATOR                       Line 16
```

```
endl                    KEYWORD                        Line 16
;                       SPECIAL SYMBOL                 Line 16
}                       SPECIAL SYMBOL                 Line 17
a                       IDENTIFIER                     Line 18
++                      OPERATOR                       Line 18
;                       SPECIAL SYMBOL                 Line 18
}                       SPECIAL SYMBOL                 Line 19
return                  KEYWORD                        Line 20
0                       INTEGER CONSTANT               Line 20
;                       SPECIAL SYMBOL                 Line 20
}                       SPECIAL SYMBOL                 Line 21


        ******** SYMBOL TABLE ********
-------------------------------------------------------------------
SNo     |       Token           |       Token Type
-------------------------------------------------------------------
1               "                       SPECIAL SYMBOL
2               %                       OPERATOR
3               (                       SPECIAL SYMBOL
4               )                       SPECIAL SYMBOL
5               0                       INTEGER CONSTANT
6               2                       INTEGER CONSTANT
7               ;                       SPECIAL SYMBOL
8               <                       OPERATOR
9               ++                      OPERATOR
10              10                      INTEGER CONSTANT
11              a                       IDENTIFIER
12              <<                      OPERATOR
13              ==                      OPERATOR
14              {                       SPECIAL SYMBOL
15              >>                      OPERATOR
16              }                       SPECIAL SYMBOL
17              if                      KEYWORD
18              Odd                     IDENTIFIER
19              cin                     PRE DEFINED FUNCTION
20              int                     KEYWORD
21              std                     KEYWORD
22              Even                    IDENTIFIER
23              endl                    KEYWORD
24              else                    KEYWORD
25              cout                    PRE DEFINED FUNCTION
26              main()                  IDENTIFIER
27              while                   KEYWORD
28              using                   KEYWORD
29              Number                  IDENTIFIER
30              return                  KEYWORD
31              namespace                       KEYWORD
-------------------------------------------------------------------
```

Phase 2: 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,while loop and for loop

Yacc tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

Context Free Grammar :-

```
%start program

%%

program

        : declaration_list;

declaration_list

        : declaration D

D

        : declaration_list

        | ;

declaration

        : variable_declaration

        | function_declaration

        | structure_definition;

variable_declaration

        : type_specifier variable_declaration_list ';'

        | structure_declaration;
```

```
variable_declaration_list
	: variable_declaration_identifier V;
V
	: ',' variable_declaration_list
	| ;
variable_declaration_identifier
	: identifier { ins(); } vdi;
vdi : identifier_array_type | assignment_operator expression ;
identifier_array_type
	: '[' initilization_params
	| ;
initilization_params
	: integer_constant ']' initilization
	| ']' string_initilization;
initilization
	: string_initilization
	| array_initialization
	| ;
type_specifier
	: INT | CHAR | FLOAT | DOUBLE
	| LONG long_grammar
	| SHORT short_grammar
	| UNSIGNED unsigned_grammar
	| SIGNED signed_grammar
	| VOID ;
unsigned_grammar
```

```
        : INT | LONG long_grammar | SHORT short_grammar | ;
signed_grammar
        : INT | LONG long_grammar | SHORT short_grammar | ;
long_grammar
        : INT | ;
short_grammar
        : INT | ;
structure_definition
        : STRUCT identifier { ins(); } '{' V1  '}' ';';
V1 : variable_declaration V1 | ;
structure_declaration
        : STRUCT identifier variable_declaration_list;
function_declaration
        : function_declaration_type function_declaration_param_statement;
function_declaration_type
        : type_specifier identifier '('  { ins();};
function_declaration_param_statement
        : params ')' statement;
params
        : parameters_list | ;
parameters_list
        : type_specifier parameters_identifier_list;
parameters_identifier_list
        : param_identifier parameters_identifier_list_breakup;
parameters_identifier_list_breakup
        : ',' parameters_list
```

```
        | ;
param_identifier
        : identifier { ins(); } param_identifier_breakup;
param_identifier_breakup
        : '[' ']'
        | ;
statement
        : expression_statment | compound_statement
        | conditional_statements | iterative_statements
        | return_statement | break_statement
        | variable_declaration;
compound_statement
        : '{' statment_list '}' ;
statment_list
        : statement statment_list
        | ;
expression_statment
        : expression ';'
        | ';' ;
conditional_statements
        : IF '(' simple_expression ')' statement conditional_statements_breakup;
conditional_statements_breakup
        : ELSE statement
        | ;
iterative_statements
        : WHILE '(' simple_expression ')' statement
```

```
        | FOR '(' expression ';' simple_expression ';' expression ')'
        | DO statement WHILE '(' simple_expression ')' ';';
return_statement
        : RETURN return_statement_breakup;
return_statement_breakup
        : ';'
        | expression ';' ;
break_statement
        : BREAK ';' ;
string_initilization
        : assignment_operator string_constant { insV(); };
array_initialization
        : assignment_operator '{' array_int_declarations '}';
array_int_declarations
        : integer_constant array_int_declarations_breakup;
array_int_declarations_breakup
        : ',' array_int_declarations
        | ;
expression
        : mutable expression_breakup
        | simple_expression ;
expression_breakup
        : assignment_operator expression
        | addition_assignment_operator expression
        | subtraction_assignment_operator expression
        | multiplication_assignment_operator expression
```

```
        | division_assignment_operator expression
        | modulo_assignment_operator expression
        | increment_operator
        | decrement_operator ;
simple_expression
        : and_expression simple_expression_breakup;
simple_expression_breakup
        : OR_operator and_expression simple_expression_breakup | ;
and_expression
        : unary_relation_expression and_expression_breakup;
and_expression_breakup
        : AND_operator unary_relation_expression and_expression_breakup
        | ;
unary_relation_expression
        : exclamation_operator unary_relation_expression
        | regular_expression ;
regular_expression
        : sum_expression regular_expression_breakup;
regular_expression_breakup
        : relational_operators sum_expression
        | ;
relational_operators
        : greaterthan_assignment_operator | lessthan_assignment_operator | greaterthan_operator
        | lessthan_operator | equality_operator | inequality_operator ;
sum_expression
        : sum_expression sum_operators term
```

```
        | term ;
sum_operators
        : add_operator
        | subtract_operator ;
term
        : term MULOP factor
        | factor ;
MULOP
        : multiplication_operator | division_operator | modulo_operator ;
factor
        : immutable | mutable ;
mutable
        : identifier
        | mutable mutable_breakup;
mutable_breakup
        : '[' expression ']'
        | '.' identifier;
immutable
        : '(' expression ')'
        | call | constant;
call
        : identifier '(' arguments ')';
arguments
        : arguments_list | ;
arguments_list
        : expression A;
```

```
A
        : ',' expression A
        | ;
constant
        : integer_constant  { insV(); }
        | string_constant   { insV(); }
        | float_constant    { insV(); }
        | character_constant{ insV(); };
%%
```

OUTPUT:

```
atul0607@atulg67:~/compiler/Phase2$ lex scanner.l
atul0607@atulg67:~/compiler/Phase2$ yacc -d parser.y
parser.y:30.1-7: warning: POSIX Yacc does not support %expect [-Wyacc]
   30 | %expect 2
      | ^~~~~~~
atul0607@atulg67:~/compiler/Phase2$ gcc lex.yy.c y.tab.c
scanner.l: In function ‘insertST’:
scanner.l:102:33: warning: implicit declaration of function ‘insertSTline’; did you mean ‘insertST’? [-Wimplicit-function-declaration]
  102 |                                 insertSTline(str1,yylineno);
      |                                 ^~~~~~~~~~~~
      |                                 insertST
scanner.l: At top level:
scanner.l:145:14: warning: conflicting types for ‘insertSTline’; have ‘void(char *, int)’
  145 |         void insertSTline(char *str1, int line)
      |              ^~~~~~~~~~~~
scanner.l:102:33: note: previous implicit declaration of ‘insertSTline’ with type ‘void(char *, int)’
  102 |                                 insertSTline(str1,yylineno);
      |                                 ^~~~~~~~~~~~
atul0607@atulg67:~/compiler/Phase2$ ./a.out < input.cpp
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------

    SYMBOL |          CLASS |        TYPE |       VALUE |    LINE NO
---------------------------------------------------------------------------------
      char |        Keyword |             |             |          4
         i |     Identifier |         int |          10 |          3
         n |     Identifier |         int |             |          3
         x |     Identifier |         int |          10 |          7
       for |        Keyword |             |             |          5
      main |     Identifier |         int |             |          2
        ch |     Identifier |        char |             |          4
     while |        Keyword |             |             |          8
        if |        Keyword |             |             |          6
       int |        Keyword |             |             |          2


                              CONSTANT TABLE
                               -------------

      NAME |           TYPE
---------------------------------------------------------------------------------
        10 | Number Constant
         0 | Number Constant
```

Phase 3: Semantic Analysis

- We wrote appropriate rules to check for semantic validity (type checking, declare before use, etc.)
- Variables are declared and can only be used in ways that are acceptable for the declared type
- We majorly focused on the grammar mentioned in the Grammar section above, we have constructed for this compiler project.
- In addition to type checking, there are other rules: new declarations don't conflict with earlier ones, access control on class fields aren't violated, break statements only appear in loops.
- Our program will be considered correct if it verifies the semantic rules and reports appropriate errors.
- Our analyzer needs to show it can handle errors related to scoping and declarations, because these form the foundation for the later work.
- We updated Symbol table with required information

Input:

```
int main()
{
   int c = 0;
   int i, a;
   for (i = 0; i < 10; i++)
   {
      a = a - i;
      int b = 450;
      b = b + a;
   }
}
```

Output:

```
atul0607@atulg67:~/compiler/Phase3$ lex ast.l
atul0607@atulg67:~/compiler/Phase3$ yacc -d ast.y
ast.y:787.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  787 |          | "INC_OP"      {       unaryop = 5;    }
      |            ^~~~~~~~
ast.y:788.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  788 |          | "DEC_OP"      {       unaryop = 6;    }
      |            ^~~~~~~~
atul0607@atulg67:~/compiler/Phase3$ gcc lex.yy.c y.tab.c
atul0607@atulg67:~/compiler/Phase3$ ./a.out < input.cpp



Symbol Table:

   SYMBOL              NAME      TYPE       SCOPE       LINE #             VALUE
 identifier               c       int         1              3              0
 identifier               i       int         1              4              0
 identifier               a       int         1              4              -
 identifier               b       int         2              8              -2147483136




Abstract Syntax Tree
                                                            main

                                                    stmt

                                            =       for

                                    i       0      ++       stmt

                                     <       i      stmt       =

main      ( stmt      ( =   i   0   )     ( for     ( ++     ( <   i   10   )   i   )     ( stmt     ( stmt     ( =   a     ( -   a   i   )   )     ( =   b   450   )
  )     ( =   b      ( +   b   a   )   )   )   )   )
```

Phase 4: Intermediate Code Generation (ICG)

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

There are different forms of output of the Intermediate Code generation phase of compiler design. These are as below:

1) 3 Address Code (implemented in our project)
2) Abstract Syntax Tree (also implemented in our project)
3) Directed Acyclic Graphs (DAGs)
4) Postfix

3 Address Code (3AC): A statement involving no more than three references (two for operands and one for result) is known as three address statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z, here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statements.

Syntax Tree: Syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by a single link in the syntax tree; the internal nodes are operators and child nodes are operands.

To generate 3-address code it is necessary to write appropriate rules in Parser that is why we majorly focused on the grammar rules written.Intermediate Code, the result of this phase is in quadruple format. The 3-address code can be represented in three different forms: Quadruples, Triples, Indirect Triples. We tried to show it in Quadruples form.

All temporary variables also get a place in the symbol table.

Input:

```
#include <iostream>
void main()
{
   int x=3,y=2;
   while(x>y){
      printf("hello world");
      x--;
   }
}
```

Output:

```
atul0607@atulg67:~/compiler/Phase4$ lex scanner.l
atul0607@atulg67:~/compiler/Phase4$ yacc -d parser.y
parser.y:66.1-7: warning: POSIX Yacc does not support %expect [-Wyacc]
   66 | %expect 1
      | ^~~~~~~
atul0607@atulg67:~/compiler/Phase4$ gcc lex.yy.c y.tab.c
atul0607@atulg67:~/compiler/Phase4$ ./a.out < input.cpp
func begin main
t0 = 3
t1 = 2
L0:
t2 = x > y
IF not t2 GoTo L1
refparam "hello world"
refparam result
call printf, 1
GoTo L0:
L1:
func end

PASSED: ICG Phase
                              PRINTING SYMBOL TABLE
                               ______________
Symbol Name |           Class |        Type |      Value |   Line No. |
________________________________________________________________________________________________________________________
         x |      Identifier |         int |          3 |          7 |
         y |      Identifier |         int |          2 |          7 |
      main |        Function |        void |            |          5 |
     while |         Keyword |             |            |          8 |
       int |         Keyword |             |            |          7 |
      void |         Keyword |             |            |          5 |
    printf |        Function |             |            |          9 |


                              PRINTING CONSTANT TABLE
                               ______________
constant name |            Type
______________________________________________________________________________________________________________
"hello world" | String Constant
         2 | Number Constant
         3 | Number Constant
```

<u>Phase 5: Code Optimization</u>

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

The optimization must be correct, it must not, in any way, change the meaning of the program.

Optimization should increase the speed and performance of the program.

The compilation time must be kept reasonable.

The optimization process should not delay the overall compiling process.

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Input:

```
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
```

Output:

```
i = 2
t0 = True
ifFalse t0 goto L0
t1 = 3
i = t1
```

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

Phase 6 – Assembly Code Generation

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

Target language: The code generator has to be aware of the nature of the target language for which the code is to be transformed. The target machine can have either CISC or RISC processor architecture.

Selection of instruction: The code generator takes Intermediate Representation as input and converts (maps) it into the target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

Register allocation: A program has a number of values to be maintained during the execution. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

Ordering of instructions: At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

This phase is used to produce target codes for three-address statements produced in the Intermediate-code generation phase.

Operations:

1. Load (from memory) (LDR Dest(Reg), Src(memloc))
2. Store (to memory) (STR Dest(memloc), Src(Reg))
3. Move (between registers) (MOV R1, R2)
4. Computations (op, dest, src1, src2)
1. ADD
2. SUB
3. MUL
4. DIV
5. Unconditional jumps (BR L)
6. Conditional jumps (Bcond R, L) cond : LZ, GZ, EZ, LEZ, GEZ, NE

Input:

```
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
t1 = a + i
a = t1
t2 = i + 1
i = t2
goto L0
L1:
t3 = 2 * a
t4 = t3 - 1
a = t4
```

Output:

```
.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]
B 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
```

RESULTS AND POSSIBLE SHORTCOMINGS

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 the 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

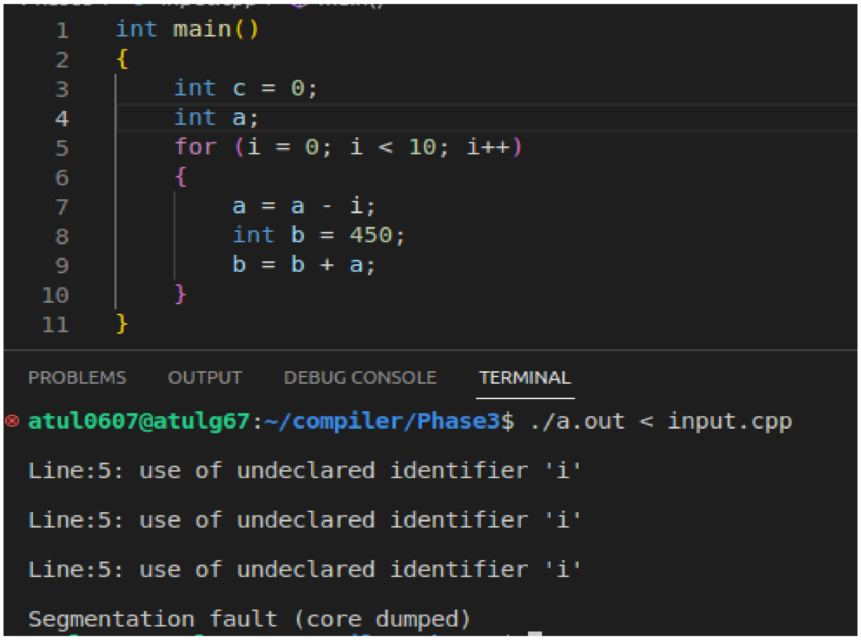
As mentioned above, we can use separate structures for the different types of tokens and then declare a union of these structures. This way, memory will be properly utilized.

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

Snapshots/errors:

Errors encountered while working on the project…

This shows the detection of an undeclared variable



```
1  int main()
2  {
3      int c = 0;
4      int a;
5      for (i = 0; i < 10; i++)
6      {
7          a = a - i;
8          int b = 450;
9          b = b + a;
10     }
11 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
atul0607@atulg67:~/compiler/Phase3$ ./a.out < input.cpp
Line:5: use of undeclared identifier 'i'
Line:5: use of undeclared identifier 'i'
Line:5: use of undeclared identifier 'i'
Segmentation fault (core dumped)
```

This shows the detection of invalid syntax at line 5

```
1  #include<iostream>
2  int main(){
3      int n,i;
4      char ch;
5      for (i=0;i<n){
6          if(i<10){
7              int x;
8              while(x<10)
9                  x++;
10         }
11     }
12 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
atul0607@atulg67:~/compiler/Phase2$ ./a.out < input.cpp
5 syntax error )
Status: Parsing Failed - Invalid
```

This shows detection of not ending the statement by ';'

```
#include <iostream>
void main()
{
    int x=3,y=2;
    while(x>y){
        printf("hello world")
        x--;
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
atul0607@atulg67:~/compiler/Phase4$ ./a.out < input.cpp
func begin main
t0 = 3
t1 = 2
L0:
t2 = x > y
IF not t2 GoTo L1
refparam "hello world"
refparam result
call printf, 1
7 syntax error x
FAILED: ICG Phase Parsing failed
```

References:

- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
- http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- http://dinosaur.compilertools.net/
- https://www.javatpoint.com/code-generation
- https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf