



Report on

C++ Mini Compiler

Submitted in partial fulfilment of the requirements for Sem VI

Compiler Design Laboratory

**Bachelor of Technology
in
Computer Science & Engineering**

Submitted by:

Mohit Gaggar	PES1201800112
Abhishek Shyam	PES1201801743
Hemanth Alva R	PES1201801937

Under the guidance of

Preet Kanwal
Associate 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 (Mini-Compiler is built for which language. Provide sample input and output of your project)	03
2.	ARCHITECTURE OF LANGUAGE: <ul style="list-style-type: none"> What all have you handled in terms of syntax and semantics for the chosen language. 	03
3.	LITERATURE SURVEY (if any paper referred or link used)	03
4.	CONTEXT FREE GRAMMAR (which you used to implement your project)	04
5.	DESIGN STRATEGY (used to implement the following) <ul style="list-style-type: none"> SYMBOL TABLE CREATION INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator). 	18
6.	IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following): <ul style="list-style-type: none"> SYMBOL TABLE CREATION INTERMEDIATE CODE GENERATION CODE OPTIMIZATION ERROR HANDLING - strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator). Provide instructions on how to build and run your program. 	19
7.	RESULTS AND possible shortcomings of your Mini-Compiler	20
8.	SNAPSHOTS (of different outputs)	21
9.	CONCLUSIONS	25
10.	FURTHER ENHANCEMENTS	25
REFERENCES/BIBLIOGRAPHY		25

INTRODUCTION

This report is on C++ Mini Compiler, which focuses on generating an intermediate code for the language for specific constructs. It works for constructs such as if-else condition and do while. The main functionality of the project is to generate an optimized intermediate code for the given C++ source code and optimized intermediate code generated.

This is done using the following steps:

- i. Lexical Analysis
- ii. Symbol Table generation
- iii. Syntax Analyzer
- iv. Semantic Analyzer
- v. Generating Intermediate Code (3- address code)
- vi. Code optimization

We have implemented various stages of the compiler using lex and yacc. PYTHON is used to optimize the intermediate code generated by the parser and generating Assembly code from intermediate code.

ARCHITECTURE OF LANGUAGE

The constructs that has been handled are:

1. Simple IF
 2. IF – ELSE Condition
 3. DO – WHILE Loop
 4. WHILE Loop
- Data Type such as int, float, char, arrays are handled.
 - Boolean expression such as <, >, <=, >=, == are handled.
 - Arithmetic expression such as +, -, *, / ,++, -- are handled.
 - Single line comments and Multi line comments have been handled.
 - Condition statements such as IF loop and IF – ELSE loop has been handled.
 - Looping Constructs such as Do – While and while loop has been handled.
 - Undeclared variables are reported has error.
 - Syntax errors are reported along with their line number.
 - If same variable is being declared twice in the same scope then they are reported has error.

LITERATURE SURVEY

- [LexAndYaccTutorial.pdf \(iitkgp.ac.in\)](http://iitkgp.ac.in) by Tom Niemann
- <https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf>
- [ANSI C grammar \(Yacc\) \(liu.se\)](http://liu.se)
- Paper on Symbol Table Implementation in Compiler Design- Dr.Jad Matta
- <http://user.it.uu.se/~kostis/Teaching/KT1-11/Slides/lecture05.pdf>
- <https://www.csie.ntu.edu.tw/~b93501005/slide5.pdf>

CONTEXT FREE GRAMMAR AND SEMANTIC ANALYSIS

```
%{
#include <stdlib.h>
#include <stdio.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "compile.c"

char* current_dtype;
char* ar = "array";
int scope=0;
int yyerror(char *msg);
struct structu{
    // double dval;
    // int ival;
    // char cval;
    char val[100];
    int type;
};
typedef struct structu structur;

struct compose
{
    node* entry;
    char id_name[100];
    int line;
};
typedef struct compose compos;
%}

%union
{
    node* entry;
    structur structure;
    compos compose;
```

```
}
```

```
%token <compose> T_IDENTIFIER T_STRING
```

```
%token T_PRINTF T_CIN T_COUT
```

```
/* Constants */
```

```
%token <val> T_FLOAT_CONSTANT
```

```
%token <val> T_DEC_CONSTANT
```

```
%token <val> T_CHAR_CONSTANT
```

```
// %token T_STRING
```

```
/* Logical and Relational operators */
```

```
%token T_LOGICAL_AND T_LOGICAL_OR T_LS_EQ T_GR_EQ T_EQ T_NOT_EQ
```

```
/* Short hand assignment operators */
```

```
%token T_MUL_ASSIGN T_DIV_ASSIGN T_MOD_ASSIGN T_ADD_ASSIGN
```

```
T_SUB_ASSIGN
```

```
%token T_LEFT_ASSIGN T_RIGHT_ASSIGN T_AND_ASSIGN T_XOR_ASSIGN
```

```
T_OR_ASSIGN
```

```
%token T_INCREMENT T_DECREMENT
```

```
/* Data types */
```

```
%token T_SHORT T_INT T_LONG T_LONG_LONG T_SIGNED T_UNSIGNED T_CONST
```

```
T_FLOAT T_CHAR
```

```
/* Keywords */
```

```
%token T_IF T_FOR T_WHILE T_DO T_CONTINUE T_BREAK T_RETURN T_MAIN
```

```
%token T_COMMA T_GG T_LL
```

```
// %type <structure> expression
```

```
// %type <structure> sub_expr
```

```
// %type <structure> constant
```

```
// %type <structure> unary_expr
```

```
// %type <structure> arithmetic_expr
```

```
// %type <structure> assignment_expr
```

```
// %type <structure> assign_op
```

```
// %type <entry> lhs
```

```
%start starter
```

```
%right '='"
```

```
%left T_LOGICAL_OR
```

```

%left T_LOGICAL_AND
%left T_EQ T_NOT_EQ
%left '<'>' T_LS_EQ T_GR_EQ
// %left '+' '-'
// %left '*' '/' '%'
%right '!'

```

```

%nonassoc T_UMINUS
%nonassoc T_LOWER_THAN_ELSE
%nonassoc T_ELSE

```

```

%%

```

```

/* Program is made up of multiple builder blocks. */

```

```

starter: starter_builder
      | builder;

```

```

/* Each builder block is either a function or a declaration */

```

```

builder: function|
      | declaration;

```

```

/* This is how a function looks like */

```

```

function: type T_IDENTIFIER '(' argument_list ')' compound_stmt
      | type T_MAIN '(' argument_list ')' compound_stmt ;

```

```

/* Now we will define a grammar for how types can be specified */

```

```

type :data_type;

```

```

data_type :sign_specifier type_specifier
      |type_specifier
      ;

```

```

sign_specifier :T_SIGNED
      |T_UNSIGNED
      ;

```

```

type_specifier :T_INT           {current_dtype = "INT";}
      |T_SHORT T_INT           {current_dtype = "SHORT";}
      |T_SHORT                 {current_dtype = "SHORT";}
      |T_LONG                   {current_dtype = "LONG";}
      |T_LONG T_INT             {current_dtype = "LONG";}
      |T_LONG_LONG              {current_dtype = "LONG_LONG";}
      |T_LONG_LONG T_INT        {current_dtype = "LONG_LONG";}

```

```

|T_FLOAT                {current_dtype = "FLOAT";}
| T_INT 'e' T_INT       {current_dtype="FLOAT";}
|T_CHAR                 {current_dtype = "CHAR";}
;

/* grammar rules for argument list */
/* argument list can be empty */
argument_list :arguments
|
;

/* arguments are comma separated TYPE ID pairs */
arguments :arguments ',' arg
|arg
;

/* Each arg is a TYPE ID pair */
arg :type T_IDENTIFIER {
    if(strcmp($2.entry->data_type,"")!=0 && strcmp($2.entry->data_type,current_dtype)!=0)
    {
        printf("Variable redeclared with different data type\n");
    }
    current_dtype="";
}
;

/* Generic statement. Can be compound or a single statement */
stmt:compound_stmt
|single_stmt
;

/* The function body is covered in braces and has multiple statements. */
compound_stmt : '{' {scope++;} statements '}' {scope--;} /*cleanup(scope+1);*/
;

statements:statements stmt
|
;

/* Grammar for what constitutes every individual statement */
single_stmt: if_block
|while_block
|do_while_block
|declaration
|assignment_expr ';'
| T_PRINTF '(' string ')' ';'
| T_CIN T_GG string_cin ';'
| T_COUT T_LL string_cout ';'

```

```

|function_call ':'
|T_RETURN ':'
|T_CONTINUE ':'
|T_BREAK ':'
|T_RETURN sub_expr ':'
| unary_expr ':'
;

string_cin: T_IDENTIFIER T_GG string_cin { /*declared($1);*/disp($1.entry);}
| T_IDENTIFIER { /*declared($1);*/disp($1.entry);}
;

string_cout: T_IDENTIFIER T_LL string_cout { /*declared($1);disp($1.entry);*/
| T_STRING T_LL string_cout
/*strcpy($1.entry->data_type,current_dtype);*/current_dtype="";disp($1.entry);}
| T_IDENTIFIER { /*declared($1);*/disp($1.entry);}
| T_STRING
/*strcpy($1.entry->data_type,current_dtype);*/current_dtype="";/*disp($1.entry);*/
;

string: T_IDENTIFIER T_COMMA string { /*declared($1);*/disp($1.entry);}
| T_STRING T_COMMA string
/*strcpy($1.entry->data_type,current_dtype);*/current_dtype="";/*disp($1);*/
| T_IDENTIFIER { /*declared($1);*/disp($1.entry);}
| T_STRING { /*strcpy($1.entry->data_type,current_dtype);*/current_dtype="";/*disp($1);*/
;

if_block:T_IF '(' expression ')' stmt %prec T_LOWER_THAN_ELSE
|T_IF '(' expression ')' stmt T_ELSE stmt
;

do_while_block: T_DO compound_stmt T_WHILE '(' expression ')' ':'

while_block: T_WHILE '(' expression ')' stmt
;

declaration:type declaration_list ':' {current_dtype="";}
;

declaration_list: declaration_list T_COMMA sub_decl
|sub_decl
;

```



```

sub_decl: init_expr
| T_IDENTIFIER {
    $1.entry=insert($1.id_name,$1.line,"0",current_dtype,scope);
    // strcpy($1.entry->data_type,current_dtype);
    // $1.entry->scope=scope;
    current_dtype="";
    display();
}

```

```

| array_index
/*|struct_block '*/
;

```

```

expression:
    expression ',' sub_expr
| sub_expr
;

```

```

sub_expr:
    sub_expr '>' sub_expr
| sub_expr '<' sub_expr
| sub_expr T_EQ sub_expr
| sub_expr T_NOT_EQ sub_expr
| sub_expr T_LS_EQ sub_expr
| sub_expr T_GR_EQ sub_expr
| sub_expr T_LOGICAL_AND sub_expr
| sub_expr T_LOGICAL_OR sub_expr
| '!' sub_expr
| arithmetic_expr
| assignment_expr
| unary_expr
;

```

```

unary_expr: T_IDENTIFIER T_INCREMENT {
    $1.entry=getval($1.id_name,scope);
    if($1.entry==NULL)
    {
        printf("Identifier %s not declared before usage in line
%d\n", $1.id_name,$1.line);
        strcpy($<structure.val>$,"0");
    }
    else
    {
        strcpy($<structure.val>$,$1.entry->value);
    }
    update_v($1.id_name,1,scope);}

```

```

|T_IDENTIFIER T_DECREMENT    {
    $1.entry=getval($1.id_name,scope);
    if($1.entry==NULL)
    {
        printf("Identifier %s not declared before usage in line
%d\n", $1.id_name,$1.line);
        strcpy($<structure.val>$,"0");
    }
    else
    {
        strcpy($<structure.val>$,$1.entry->value);
    }
    update_v($1.id_name,0,scope);}
|T_DECREMENT T_IDENTIFIER    {
    update_v($2.id_name,0,scope);
    $2.entry=getval($2.id_name,scope);
    if($2.entry==NULL)
    {
        printf("Identifier %s not declared before usage in line
%d\n", $2.id_name,$2.line);
        strcpy($<structure.val>$,"0");
    }
    else
    {
        strcpy($<structure.val>$,$2.entry->value);
    }
}
|T_INCREMENT T_IDENTIFIER    {update_v($2.id_name,1,scope);

    $2.entry=getval($2.id_name,scope);
    if($2.entry==NULL)
    {
        printf("Identifier %s not declared before usage in line
%d\n", $2.id_name,$2.line);
        strcpy($<structure.val>$,"0");
    }
    else
    {
        strcpy($<structure.val>$,$2.entry->value);
    }
}
;

```

```

init_expr:T_IDENTIFIER assign_op arithmetic_expr
{
    $1.entry=insert($1.id_name,$1.line,$<structure.val>3,current_dtype,scope);
    current_dtype="";
}

```

```

        // $<compose>$.entry = $1.entry;
        // $<compose>$.line = $1.line;
        // strcpy($<compose>$.id_name,$1.id_name);

        // strcpy($<structure.val>$,$<structure.val>3);
        display();
    }
    |T_IDENTIFIER assign_op unary_expr
    {$1.entry=insert($1.id_name,$1.line,$<structure.val>3,current_dtype,scope);
        current_dtype="";
    update_val($1.id_name,$<structure.val>3,scope);display();}

assignment_expr :T_IDENTIFIER assign_op arithmetic_expr {

    // printf("In assignment expr\n");
    update_val($1.id_name,$<structure.val>3,scope);

    // $<compose>$.entry = $1.entry;
    // $<compose>$.line = $1.line;
    // strcpy($<compose>$.id_name,$1.id_name);

    // strcpy($<structure.val>$,$<structure.val>3);

    /*
        $1.entry=insert($1.id_name,$1.line,$<structure.val>3,current_dtype,scope);
        strcpy($1.entry->data_type,current_dtype);
        $1.entry->scope=scope;
        // current_dtype="";
        // printf("IN assignment_expr=%s\n\n", $<structure.val>3);
    */
    display();
}
    |T_IDENTIFIER assign_op unary_expr
    {update_val($1.id_name,$<structure.val>3,scope);}
    |unary_expr assign_op unary_expr
    ;

assign_op: '='
    |T_ADD_ASSIGN
    |T_SUB_ASSIGN
    |T_MUL_ASSIGN
    |T_DIV_ASSIGN
    |T_MOD_ASSIGN
    ;

```

```
arithmetic_expr: E {strcpy($<structure.val>$,$<structure.val>1); }
```

```
E: E '+' T {
    $<structure.type>$=function_checktype(&$<structure>1,&$<structure>3);

    if($<structure.type>$==1){
        printf("Incompatible data_type %s %s",$<structure.val>1,$<structure.val>3);
    }
    else if($<structure.type>$ == 2){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)+atoi($<structure.val>3));
    }
    else if($<structure.type>$ == 3){

sprintf($<structure.val>$,"%f",atof($<structure.val>1)+atof($<structure.val>3));
    }
}
```

```
| E '-' T { $<structure.type>$=function_checktype(&$<structure>1,&$<structure>3);

    if($<structure.type>$==1){
        printf("Incompatible data_type %s %s",$<structure.val>1,$<structure.val>3);
    }
    else if($<structure.type>$ == 2){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)-atoi($<structure.val>3));
    }
    else if($<structure.type>$ == 3){

sprintf($<structure.val>$,"%f",atof($<structure.val>1)-atof($<structure.val>3));
    }
}
```

```
| T {

strcpy($<structure.val>$,$<structure.val>1);$<structure.type>$=$<structure.type>1;}

;
```

```
T: T '*' F {

    //printf("%d is TYPE OF $$ \n",$<structure.type>$);
```

```

    $<structure.type>$=function_checktype(&$<structure>1,&$<structure>3);

    if($<structure.type>$==1){
        printf("Incompatible data_type %s %s",$<structure.val>1,$<structure.val>3);
    }
    else if($<structure.type>$ == 2){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)*atoi($<structure.val>3));
    }
    else if($<structure.type>$ == 3){

sprintf($<structure.val>$,"%f",atof($<structure.val>1)*atof($<structure.val>3));
    }
}
| T '/' F { $<structure.type>$=function_checktype(&$<structure>1,&$<structure>3);

    if($<structure.type>$==1){
        printf("Incompatible data_type %s %s",$<structure.val>1,$<structure.val>3);
    }
    else if($<structure.type>$ == 2){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)/atoi($<structure.val>3));
    }
    else if($<structure.type>$ == 3){

sprintf($<structure.val>$,"%f",atof($<structure.val>1)/atof($<structure.val>3));
    }
}
| T '%' F { $<structure.type>$=function_checktype(&$<structure>1,&$<structure>3);

    if($<structure.type>$==1){
        printf("Incompatible data_type %s %s",$<structure.val>1,$<structure.val>3);
    }
    else if($<structure.type>$ == 2){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)%atoi($<structure.val>3));
    }
    else if($<structure.type>$ == 3){

sprintf($<structure.val>$,"%d",atoi($<structure.val>1)%atoi($<structure.val>3));
    }
}
| F {
    strcpy($<structure.val>$,$<structure.val>1);$<structure.type>$=$<structure.type>1;}

;

```

```

F: T_IDENTIFIER {$1.entry=getval($1.id_name,scope);
    if($1.entry==NULL)
    {
        printf("Identifier %s not declared before usage in line
%d\n", $1.id_name, $1.line);
        strcpy($<structure.val>$, "0");
    }
    else
    {
        strcpy($<structure.val>$, $1.entry->value);
        //$<structure.type>$=$1.entry->type;
        if(strcmp("INT", $1.entry->data_type)==0)
        {
            $<structure.type>$ = 2;
        }
        else if(strcmp("FLOAT", $1.entry->data_type)==0)
        {
            $<structure.type>$ = 3;
        }
        else
        {
            $<structure.type>$ = 1;
        }
    }
    // strcpy($1.entry->data_type, current_dtype);
    // $1.entry->scope=scope;

    //strcpy($<structure.val>$, $<structure.val>1);
    //$<structure.type>$=$<structure.type>1; display();

}
| unary_expr
{strcpy($<structure.val>$, $<structure.val>1); $<structure.type>$=$<structure.type>1;} /* 28
r/r conf here*/
| constant
{strcpy($<structure.val>$, $<structure.val>1); $<structure.type>$=$<structure.type>1;}
| '(' E ')'
{strcpy($<structure.val>$, $<structure.val>2); $<structure.type>$=$<structure.type>1;}
| '.' F
{strcpy($<structure.val>$, "-"); strcat($<structure.val>$, $<structure.val>2); $<structure.type>$
=$<structure.type>1;}
;

constant: T_DEC_CONSTANT {strcpy($<structure.val>$, $<structure.val>1);}
| T_FLOAT_CONSTANT {strcpy($<structure.val>$, $<structure.val>1);}
| T_CHAR_CONSTANT {strcpy($<structure.val>$, $<structure.val>1);}

```

```

;

array_index: T_IDENTIFIER subhub ';' {$1.entry=insert($1.id_name,$1.line,"ARR
VAI",current_dtype,scope);
                current_dtype="";
                display();}

;

subhub : subhub '[' sub_expr ']' { char * ns = (char*)malloc(sizeof(strlen(current_dtype)+8));
    strcpy(ns,current_dtype);
    strcat(ns," ARRAY");
    current_dtype=ns;
}
| '[' sub_expr ']' {
    char * ns = (char*)malloc(sizeof(strlen(current_dtype)+8));
    strcpy(ns,current_dtype);
    // printf("after strcpy\n");
    strcat(ns," ARRAY");
    // printf("len ns len current_dtype %d
%d\n",strlen(ns),strlen(current_dtype));
    // printf("after strcat\n");
    current_dtype=ns;
    // printf("after strcpy ns %s\n",current_dtype);
}

/*
{

char * ns = (char*)malloc(sizeof(strlen(current_dtype)+8));
strcpy(ns,current_dtype);
strcat(ns," ARRAY");
current_dtype=ns;
}
*/
;

function_call: T_IDENTIFIER '(' parameter_list ')'
|T_IDENTIFIER '(' ')'
;

parameter_list:
    parameter_list T_COMMA parameter
|parameter
;

parameter: sub_expr
|T_STRING

;

```

```
%%
```

```
#include "lex.yy.c"  
#include <ctype.h>
```

```
int function_checktype(structur * a,structur * b){  
  
    // printf("INSIDE FUNC %d%d%s%s\n",a->type,b->type,a->val,b->val);  
    if(a->type==1 && b->type==2){  
        a->type=2;  
        // a->val=atoi(a->val);  
        return 2;  
    }  
    if(b->type==1 && a->type==2){  
        b->type=2;  
        // b->val=atoi(b->val);  
        return 2;  
    }  
  
    if(a->type==2 && b->type==3){  
        a->type=3;  
        return 3;  
    }  
    if(b->type==2 && a->type==3){  
        b->type=3;  
        return 3;  
    }  
  
    if(a->type==b->type && a->type==1){  
        a->type=2;  
        // a->val=atoi(a->val);  
        b->type=2;  
        // b->val=atoi(b->val);  
        return 2;  
    }  
    if(a->type==b->type ){  
        return a->type;  
    }  
  
    else{  
        return -1;  
    }  
}
```

```
int main(int argc, char *argv[])
```



```

{
    // symbol_table = create_table();
    // constant_table = create_table();

    yyin = fopen(argv[1], "r");

    if(!yyparse())
    {
        printf("\nParsing complete\n");
    }
    else
    {
        printf("\nParsing failed\n");
    }

    // printf("\n\tSymbol table");
    display();

    fclose(yyin);
    return 0;
}

int yyerror(char *msg)
{
    printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
}

```

DESIGN STRATEGY

1.) Symbol Table:

A structure is maintained to keep track of all the variables, constants, operators and keywords in the input. Symbol Table is used to store information about the occurrences of various entries. Expressions are evaluated and the values of the variables are updated in the symbol table.

Information stored in Symbol table:

- Variable Name
- Value
- Data Type
- Scope Number
- Line Number

At the end of parsing, the updated symbol table is displayed. Symbol table is used to check the scope, type of the variable and also verify if the variable has been declared.

2.) Intermediate code generation:

Intermediate code tends to be a machine independent code. Implementation of the intermediate code generator is done using stacks, which is created by us instead of the stack provided by yaccs.

A statement involving no more than 3 references is known as the three address statement.

A sequence of three address statements is known as three-address code. Three-address code is an abstract form of intermediate code.

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

3.) Code Optimization:

This phase consists of the various machine independent code optimizations techniques applied onto the Intermediate code generated (ICG). The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values

The following are the optimizations performed- constant propagation and constant folding in sequential blocks followed by dead code elimination.

4.) Error Handling:

Error handling is being done from yaccs file. Whenever the parser encounters an error, it prints the line number and the type of error. The compiler supports and implements multi-line error handling.

This compiler supports all syntax related errors. On the lexical side we do not handle size more than 30. It also generates an error when there's any illegal character (when the input doesn't match with regex expression). This compiler supports and implements semantic error handling. It checks for variable redeclaration when the scope of the variable is same. It also handles incompatible types of operand and undeclared variables.

All comments are removed from code before parsing.

IMPLEMENTATION DETAILS

1.) Symbol Table:

Data structure used in the symbol table is struct. This compiler parses the code line by line, and whenever it encounters a variable declaration or assignment operation, the symbol table gets updated with the Variable Name, Line Number, Scope number, latest Value and Data Type. A variable name can be found in different scopes and a variable name with different scope can have different values. This compiler consists of different functions to insert variables onto the symbol table and update the values of the variables, to check if a variable is present in the symbol table and extract the value of the variable for expression evaluation.

2.) Intermediate Code Generation:

The Data structure used to implement the intermediate code generator is a stack. Whenever an assignment operator is encountered, the operator and operand is pushed onto the stack. While printing, the top of the stack is popped and is displayed as in three-address code.

An important concept used in implementing Intermediate Code is Back-patching. The data structure used to represent Three-address code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2 and result. All the three-address code is stored as an array of Quadruple.

3.) Code Optimization:

We have implemented code optimization in python. For dead code elimination we continuously loop through the code until there is no code that can be eliminated. The techniques used in this phase are:

Constant Folding and Constant Propagation.

- Constant Folding: If any expression is encountered, it first checks if both the operands are constants and then the expression gets evaluated.
- Constant Propagation: On expression evaluation, if the expression has at least one operand as a variable then it extracts the value from the table and later constant folding is applied to these substituted values.
- Dead code removal

4.) Error Handling:

Error key handling is done in the yaccs file, in our compiler it is done with the help of self defined functions instead of YYERROR function of yaccs. Whenever the compiler detects an error, instead of terminating the program, the compiler continues to read the instructions and print the line numbers of those errors.

In the semantic phase of the compiler variable redeclaration (when the scope is the same), Undeclared variables and type mismatch has been handles. It also handles type mismatch during expression evaluation. For Type mismatch, we check the data type of the value and datatype of the variable in the symbol table, if it doesn't match it throws an error.

RESULTS AND SHORTCOMINGS

The result achieved is that we have a mini compiler and parses the grammar corresponding to C++ code (IF-ELSE condition and Do-While loop) and finally generate an optimized code.

This mini compiler works as any other compiler, on the provided constraint. It efficiently compiles the input file and identifies and displays any error present in the code.

There are a few shortcomings in our implementation; one among them is that 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. It works well only in sequential programs. This needs to be rectified.

It is not capable of handling each and every construct of the C++ language as it currently stands, and will require significant changes to do so.

SNAPSHOTS

Sample INPUT:

```
#include<stdio.h>
void main()
{
    int a=0;
    int b = a*3;
    while( a > b ){
        a = a+1;
    }
    if( b < = c ){
        a = 10;
    }
    else{
        a = 20;
    }

    if( b > = 0 )
    {
        b=20;
    }
    else
    {
        b=30;
    }
    a = 100;
}
```

Sample OUTPUT: (ICG)

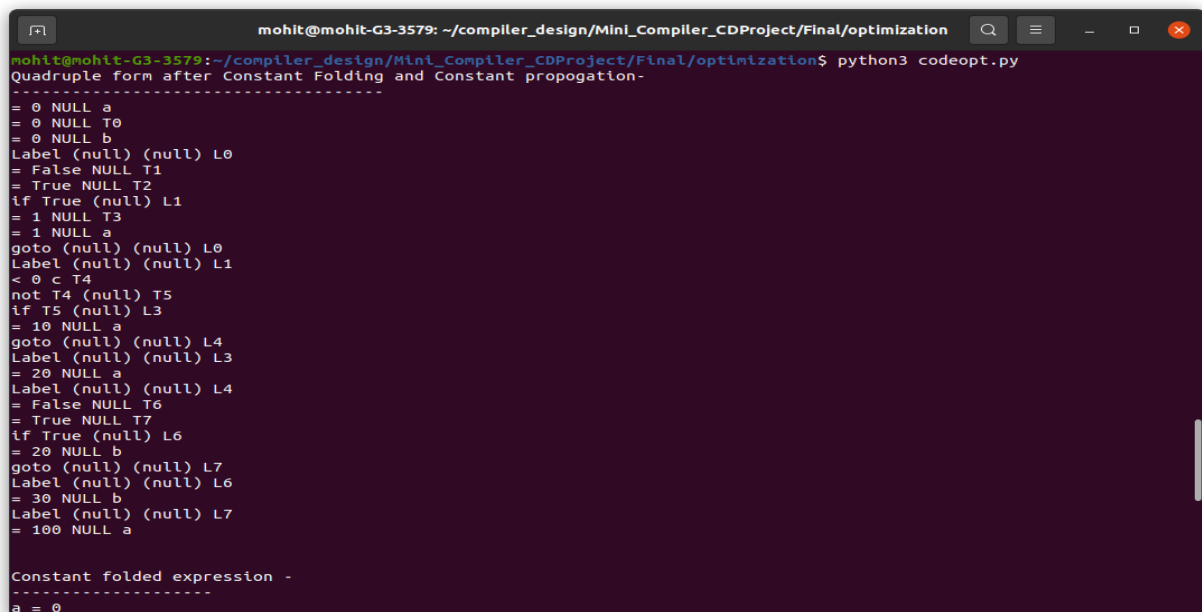
```
hemanthalva@hemanthalva:~/Mini_Compiler_CDProject/Final/ICG$ ./a.out < test_cases/input.c
a = 0
T0 = a * 3
b = T0
L0:
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b <= c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
T6 = b >= 0
T7 = not T6
if T7 goto L6
b = 20
goto L7
L6:
b = 30
L7:
a = 100
Input accepted.
Parsing Complete
```

```
-----Quadruples-----
Operator      Arg1      Arg2      Result
=             0         (null)    a
*             a         3         T0
=             T0        (null)    b
Label         (null)    (null)    L0
>             a         b         T1
not           T1        (null)    T2
if            T2        (null)    L1
+             a         1         T3
=             T3        (null)    a
goto          (null)    (null)    L0
Label         (null)    (null)    L1
<=            b         c         T4
not           T4        (null)    T5
if            T5        (null)    L3
=             10        (null)    a
goto          (null)    (null)    L4
Label         (null)    (null)    L3
=             20        (null)    a
Label         (null)    (null)    L4
>=            b         0         T6
not           T6        (null)    T7
if            T7        (null)    L6
=             20        (null)    b
goto          (null)    (null)    L7
Label         (null)    (null)    L6
=             30        (null)    b
Label         (null)    (null)    L7
=             100       (null)    a
```

Sample INPUT:

```
=          0          (null)      a
*          a          3          T0
=          T0         (null)      b
Label      (null)     (null)      L0
>          a          b          T1
not        T1         (null)      T2
if         T2         (null)      L1
+          a          1          T3
=          T3         (null)      a
goto       (null)     (null)      L0
Label      (null)     (null)      L1
<=         b          c          T4
not        T4         (null)      T5
if         T5         (null)      L3
=          10         (null)      a
goto       (null)     (null)      L4
Label      (null)     (null)      L3
=          20         (null)      a
Label      (null)     (null)      L4
>=         b          0          T6
not        T6         (null)      T7
if         T7         (null)      L6
=          20         (null)      b
goto       (null)     (null)      L7
Label      (null)     (null)      L6
=          30         (null)      b
Label      (null)     (null)      L7
=          100        (null)      a
```

Sample OUTPUT [Code optimization]:



```
mohit@mohit-G3-3579: ~/compiler_design/Mini_Compiler_CDProject/Final/optimization
mohit@mohit-G3-3579:~/compiler_design/Mini_Compiler_CDProject/Final/optimization$ python3 codeopt.py
Quadruple form after Constant Folding and Constant propagation-
-----
= 0 NULL a
= 0 NULL T0
= 0 NULL b
Label (null) (null) L0
= False NULL T1
= True NULL T2
if True (null) L1
= 1 NULL T3
= 1 NULL a
goto (null) (null) L0
Label (null) (null) L1
< 0 c T4
not T4 (null) T5
if T5 (null) L3
= 10 NULL a
goto (null) (null) L4
Label (null) (null) L3
= 20 NULL a
Label (null) (null) L4
= False NULL T6
= True NULL T7
if True (null) L6
= 20 NULL b
goto (null) (null) L7
Label (null) (null) L6
= 30 NULL b
Label (null) (null) L7
= 100 NULL a

Constant folded expression -
-----
a = 0
```


CONCLUSION

In Conclusion, a mini-compiler was implemented, in addition to the specified constructs, while loop and other minor constructs have been implemented. This compiler handles the errors and also prints the corresponding line number. It is written in python for code optimization. The tools used for building this are the Lex and Yacc.

FURTHER ENHANCEMENTS

For our future work, we would make the script even more robust as to handle all aspects of C++ language and make it more efficient. 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.

REFERENCES

[Intermediate code generation using LEX & YACC for Control Flow and Switch Case statements \(professionalcipher.com\)](http://professionalcipher.com)

[Lex - A Lexical Analyzer Generator \(compilertools.net\)](http://compilertools.net)

[LexAndYaccTutorial.pdf \(iitkgp.ac.in\)](http://iitkgp.ac.in)

[ANSI C grammar \(Yacc\) \(liu.se\)](http://liu.se)

[final.codegen.pdf \(ucdavis.edu\)](http://ucdavis.edu)