Report on

"C++ mini-compiler designed in C"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Rishabh Jain	**PES1201700099**
Jeevana R Hegde	**PES1201700633**
Salman Ahmed	**PES1201700954**

Under the guidance of

Prakash C O
Assistant Professor
PES University, Bengaluru

January – May 2020

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

INTRODUCTION

In our project, a mini-compiler for the language C++ has been built. lex and yacc have been used which are tools in the language C to achieve this. The constructs handled in our mini-compiler are **for loops, do-while loops** and **if conditions.**

All the phases of a compiler have been implemented in this project, which are :

- ***Lexical phase*** – Lex reads the input file and generates tokens.
- ***Syntax phase*** – Yacc parses through the tokens returned by lex and ensures that the input file follows the correct syntax of the programming language.
- ***Semantic phase*** – Actions are called after grammar productions to verify the correctness of the meaning of the statements, to ensure they are semantically correct.
- ***Intermediate code generation*** – Three address code is generated for the input file.
- ***Code optimization*** – The three address code generated is optimized so as to speed up the execution of the program. The techniques implemented are constant propagation, copy propagation and dead code elimination.
- ***Assembly Code Generation*** – The optimized code is converted into assembly code.

Our sample input and output is shown below.

Sample Input:

```
int main()
{
    int a,b=0;

    a=5;
    int c=b;
    for(int i=0;i<a;i=i+1)
    {
        b=1;
    }

    float z1=4.2,z2=4.8;


    do{
        b = b+2;
        c = 10;
        if(b == 5)
        {
            break;
        }
    }while(c<a);

    if(c<b)
    {
        if(a<b)
        {
            a = 10;
            cout<<a;
        }
    }
    int d = a/b+c;
```

Sample Output(Assembly Language)

```
MOV R0, #0
ST b, R0
MOV R1, #5
ST a, R1
MOV R2, #0
ST c, R2
MOV R3, #0
ST i, R3
L0: SUB R4, R3, R1
BGEZ R4, L1
MOV R0, #1
ST b, R0
MOV R5, #1
ADD R3, R3, R5
ST i, R3
BR L0
L1: L2: MOV R6, #2
ADD R0, R0, R6
ST b, R0
MOV R2, #10
ST c, R2
MOV R7, #5
SUB R8, R0, R7
BNE R8, L4
BR L3
L4: SUB R9, R2, R1
BLTZ R9, L2
L3: MOV R10, #10
SUB R11, R10, R0
BGEZ R11, L5
SUB R12, R7, R0
BGEZ R12, L6
L6: L5:
```

ARCHITECTURE OF LANGUAGE

Both syntactical and semantic errors have been taken care of in our language. The implementation of the same is shown in later on sections.

Syntax errors handled:

- Missing semicolon
- Right bracket missing
- Left bracket missing

Semantic errors handled:

- Usage of a variable not declared
- A variable being declared multiple times.

Literature Survey

The various links used for our project are:

- https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- https://2k8618.blogspot.com/2011/06/intermediate-code-generator-for-if-then.html?m=0

A numerous number of PDF's provided by our professors helped in the understanding of the representations required in the different phases of the compiler.

CONTEXT FREE GRAMMAR

```
program_beginning   : PRO_BEG left_brac right_brac left_brac compound_statement
                      right_brac
                      ;

declarator_list     : value ',' declarator_list
                    | value '=' arit_expression
                    | value '=' arit_expression ',' declarator_list
                    | value
                    ;

compound_statement: statement compound_statement
                    | statement
                    ;

statement           : exp_statement
                    | selection_statement
                    | iteration_statement
                    | jump_statement semi
                    ;

exp_statement       : expression semi
                    ;

selection_statement : IF left_brac expression right_brac left_brac compound_statement
                      right_brac
                      ;

iteration_statement : FOR left_brac for_init_statement semi for_condition semi
                      for_updation ')' left_brac compound_statement right_brac
                    | DO left_brac compound_statement right_brac WHILE left_brac
                      rel_expression right_brac semi
                      ;

jump_statement      : BREAK
                    | CONTINUE
                    | RETURN expression
                    ;
```

```
expression          : rel_expression
                    | value '=' arit_expression
                    | TYPE_SPEC declarator_list
                    | print
                    | arit_expression
                    ;

rel_expression      : arit_expression LT arit_expression
                    | arit_expression GT arit_expression
                    | arit_expression LE arit_expression
                    | arit_expression GE arit_expression
                    | arit_expression EQ arit_expression
                    | arit_expression NE arit_expression
                    ;

arit_expression     : arit_expression1
                    | arit_expression '+' arit_expression
                    | arit_expression '-' arit_expression
                    ;

arit_expression1    : value
                    | arit_expression1 '*' arit_expression1
                    | arit_expression1 '/' arit_expression1
                    ;

for_init_statement  : for_init_st_val ',' for_init_statement
                    | for_init_st_val
                    |
                    ;

for_init_st_val     : TYPE_SPEC value '=' arit_expression
                    | value '=' arit_expression
                    ;

for_condition       : rel_expression
                    |
                    ;

for_updation        : for_updation_val ',' for_updation
                    | for_updation_val
                    |
                    ;
```

```
for_updation_val    : value '=' arit_expression
                    ;

print               : COUT REDIR_OPER arit_expression print1
                    ;

print1              : REDIR_OPER arit_expression print1
                    |
                    ;

value               : ID
                    | INT_CONS
                    | FLOAT_CONS
                    |STRING_CONS
                    |CHAR_CONS
                    ;

semi                : ';'
                    | error
                    ;

left_brac           : '('
                    | '{'
                    | error
                    ;

right_brac          : ')'
                    | '}'
                    | error
                    ;
```

DESIGN STRATEGY

I. Symbol Table Creation

In order to implement the symbol table, the details of each variable created in a data structure, a linked list is to be stored and the data structure is to be updated continuously as we parsed through the program. After completely parsing through the input file containing our C++ program, the details of our symbol table is written onto a text file.

For every variable that is present in our input file, a node is created and it is stored in the structure for the symbol table. Semantic checks are performd to ensure that the variable has been declared before it is being used. In case a variable hasn't been declared, it won't be written into the final symbol table. All variables are initially given the value 0 and their value gets updated as the parsing continues. Many intermediate variables are used to keep track of additional information required in the symbol table such as line number, scope, etc. A new node is also created for a variable every time it is used in a different scope.

II. Abstract Syntax Tree

A common data structure has been created which can store the details of any statement in our program. As and when the grammar is parsed, for every grammar production that has been completed a node is created and stored in a stack. This is done in case another grammar production requires the node that was just created. At the end of every statement the node is popped from the stack and stored in the tree.

It is started off with a root node and every node created following that will be a branch of this node. After the input has been completely scanned, the tree is printed from the root node branch by branch till the complete tree has been printed.. Each branch of the tree is printed in the order it appears in the input file.

III. Intermediate Code Generation

A three-address code has been used to implement intermediate code generation. After each grammatical production (or after every expression), a function specific to that production is present. This is meant to create the three-address code for that statement. The function then writes the statement into a text file.

A variable is present that continuously updates itself so as to create new intermediate variables as and when we require them, and have implemented a similar feature for creating new branches. Stacks are used to keep track of the variables required in the instructions of the three address code. Information about every single statement is stored onto a structure which acts as a quadruple. The quadruple is used to keep the entire intermediate code.

IV. Code Optimization

The intermediate code generated is taken and modified and stored in a new data structure which is then printed onto a text file. After generating the instruction for the three-address code, it is modified in order to be optimized. The different optimizations which are performed are:

- ***Constant propagation*** – If any variable which has been initialized to a constant value is noticed, it is replaced with the value of the variable with that constant value in the expression it is being used. Thus, this expression won't need to be evaluated at run time.
- ***Copy propagation*** – All the occurrences of targets of direct assignments are replaced with their values. So, if variable1 has been assigned the value of another variable (variable2), then variable2 will replace variable1 in any expression variable1 appears in (unless its value is modified).
- ***Dead code elimination*** – Any unnecessary instruction is eliminated in the three-address code to reduce the size of the code and eventually reduce the run time. All unnecessary assignments are eliminated.

In order to achieve these optimizations, various data structures are used. Separate lists are present for variables which have been assigned a constant value or another single variable, to replace the variable in any other expression it is used.
In order to achieve dead code elimination, the line numbers corresponding to when the variable has been assigned a value in a data structure is stored and checked to see if the variable has been used after that. If it hasn't, then removing the instructions corresponding to that assignment from the intermediate code will not affect the result of the of the program and hence they are eliminated.

V. Error Handling

Error handling has been taken care of by our context free grammar and a couple of user-defined functions. The syntactic errors, such as a missing semicolon or missing bracket have been handled by adding new production rules corresponding to the type of error, which place a call to the yyerror function. After it comes out of the yyerror function, the yacc program continues parsing from where it left off.

In order to handle a variable being used when it isn't declared or a variable being declared multiple times, we have created two functions which can check that. Every time a variable is used in our input program, we check to see if it has been declared before by placing a call to the required functions. This is done by storing each variable in a structure with another member in that structure telling us if the variable has been declared or not. In case of any discrepancy, yyerror will be called, and then yacc continues parsing through the input file.

VI. Target Code Generation

The target code is generated from the optimized intermediate code created. Two lists are maintained to store the registers, one for all the used ones and one for the available registers. Every statement in the optimized intermediate code is converted to the respective assembly language instruction.

Each time a register is required for a variable or maybe for a constant is being used, the existence of the variable is checked in the already existing list of sorted registers. If it exists, there is no need to create an extra instruction of loading the value of the variable from memory into the register. If not, the next available register is taken and the variable's value is loaded from the memory into that register. If no register is available, the last used register is found and used. This is easily obtained because every time a register is used, it moves to the back of the used registers list. Hence the first register in the list is always the register that was least recently used. Also, since the intermediate code contains many intermediate variables which won't be used in the rest of the instructions, each time the list of available registers is empty, all the registers corresponding to the intermediate variables are freed.

The instructions LD and MOV are used to load a variable into a register and move the value of a constant into a register respectively. The arithmetic instructions used are ADD, SUB, MUL and DIV based on the operation being performed. Branching instructions based on conditions have also been used, such as BLTZ, BGEZ, BNE, etc.

IMPLEMENTATION DETAILS

I. Symbol Table Creation

A structure which is implemented as a linked list is used to store the symbol table.

```
typedef struct node
{
    char id_name[20];
    char id_type[20];
    int scope;
    int val_int;
    int storage_req;
    float val_float;
    int line_no[100];
    int no_of_lines;
    struct node *link;
    int has_been_declared;
}node;
node *head = NULL;
```

The function shown below adds variables or updates variables in the symbol table.

```
void update_st(char *var)
{
    if(head == NULL)
    {
        head = (node *)malloc(sizeof(node));
        strcpy(head->id_name,var);
        strcpy(head->id_type,prev_type);
        head->scope = scope;
        head->val_int = 0;
        head->val_float = 0;
        head->no_of_lines = 1;
        head->line_no[0] = line_no;
        head->link = NULL;
        if((strcmp(head->id_type,"int") == 0) || (strcmp(head->id_type,"float") == 0))
        {
            head->storage_req = 4;
        }
        else if(strcmp(head->id_type,"char") == 0)
        {
            head->storage_req = 1;
        }
        if(line_of_declaration == line_no  && this_var_is_being_declared == 1)
        {
            head->has_been_declared = 1;
        }
        else
        {
            head->has_been_declared = 0;
        }
        return;
    }
```

II. Abstract Syntax Tree

The structure shown below is used to store each node of the syntax tree. It consists of an enum and a union of structures. The enum is used to specify the statement the node is going to hold and in the union of structures there is a structure defined for each type of statement in the input file.

```
typedef struct AST
{
    enum
    {
        main_func, arit_expression, rel_expression,
        assign_expression, declaration,
        for_loop, do_while_loop, if_cond,
        jump_stat, comp_stat, print_stat,
        var, str_cons, float_cons, int_cons
    } tag;
    union
    {
        struct
        {
            struct AST* body;
        } main_body;

        struct
        {
            char oper[3];
            struct AST* left;
            struct AST* right;
        } arit;

        struct
        {
            char oper[3];
            struct AST* left;
            struct AST* right;
        } rel;
```

A function to create a node for each type of statement and insert the node into the tree has been defined.

```
// AST
void push_cs_onto_stack(ast *comp_statement);
ast* pop_cs_from_stack();

void add_stat_to_cs_stack();

ast* create_node_main_init();
void create_node_arit_init(char *oper);
void create_node_rel_init(char *oper);
void create_node_assign_init(char *oper);
void create_node_decl_init();
void create_node_for_init();
void create_node_dowhile_init();
void if_init();
void jump_stat_init(char *type);
ast* comp_stat_init();
void print_stat_init(char *oper);
void var_init(char *var_name);
void str_val_init(char *val);
void float_init(float val);
void int_init(int val);

void push_onto_valarit_stack(ast *var);
ast* pop_from_valarit_stack();

void write_ast_into_file();
void writing_ast(FILE *f_ast, int tab, ast *node);
void print_tab(FILE *f_ast, int tab);
```

The functions shown below are used to write the entire tree into a text file.

```
void write_ast_into_file()
{
    FILE *f_ast = fopen("ast.txt","w");
    int no_of_tab_spaces = 0;

    writing_ast(f_ast,no_of_tab_spaces,head_ast);

    fclose(f_ast);
}

void writing_ast(FILE *f_ast, int tab, ast* node)
{
    if(node->tag == main_func)
    {
        fprintf(f_ast,"<Main Tree>\n");
        tab += 1;
        writing_ast(f_ast,tab,node->op.main_body.body);
        tab -= 1;
        fprintf(f_ast,"<Main Tree Closed>\n");
    }

    else if(node->tag == arit_expression)
    {
        print_tab(f_ast,tab);
        fprintf(f_ast,"<Arit Exp>\n");

        tab += 1;
        print_tab(f_ast,tab);
        fprintf(f_ast,"<Arit Exp oper> %s\n",node->op.arit.oper);
```

III. Intermediate Code Generation

A stack has been used to add variables when read from the input file and are popped when they are required in an instruction. Other stacks are also used to store the branch to jump to for a later instruction or to store the updation value of a for loop. A structure is also created for quadruples, which has 4 fields such that any three-address code instruction can be stored in it.

```
// ICG
int inter_var_no = 0;
int branch_no = 0;

char icg_var_stk[100][20];
int top = -1;

int branch_stk[50];
int top_br = -1;

int is_for_upd = 0;
char for_upd_val[10][50];
int top_for = -1;

int loop_br[20];
int top_loop = -1;

char do_while_br[10][20];
int top_dow = -1;


FILE *f_icg;

typedef struct quadruples
{
    char op[20];
    char arg1[20];
    char arg2[20];
    char res[20];
    struct quadruples *link;
} quad;
quad *headq = NULL;
```

A function has been defined for each type of expression which takes care of creating the three-address code instruction, writing it into a text file and storing it in the quadruples structure.

```
// ICG
void push_onto_icg_stack(char val[20]);
void pop_from_icg_stack(char val[20]);
void create_inter_var(char inter[20]);
void create_new_branch_var(char branch[20]);

void assign_icg();
void arit_icg();
void rel_icg();
void create_branch();
void rel_expr();
void for_branch_dec();
void if_branch_dec();
void rel_expr_dowhile();
void do_while_after_branch();
void do_while_print_after_br();
void break_icg();
void print_icg();

void insert_into_quad(char op[20], char arg1[20], char arg2[20], char res[20]);
void write_quad(FILE *f_quad);
```

The first function below writes data into the quadruples data structure. The second function writes every single node of the quadruples structure into a table with the representation of every single instruction present.

```
void insert_into_quad(char op[20], char arg1[20], char arg2[20], char res[20])
{
    quad *temp = (quad *)malloc(sizeof(quad));
    strcpy(temp->op,op);
    strcpy(temp->arg1,arg1);
    strcpy(temp->arg2,arg2);
    strcpy(temp->res,res);

    if(headq == NULL)
    {
        headq = temp;
    }
    else
    {
        quad *temp1 = headq;
        while(temp1->link)
        {
            temp1 = temp1->link;
        }
        temp1->link = temp;
    }
}

void write_quad(FILE *f_quad)
{
    quad *temp = headq;
    fprintf(f_quad,"%s\t\t%20s\t\t%20s\t\t%20s\t\t%20s\n\n","Pos","Op","Arg1","Arg2","Res");
    int pos = 1;
    while(temp)
    {
        fprintf(f_quad,"%d\t\t%20s\t\t%20s\t\t%20s\t\t%20s\n",pos,temp->op,temp->arg1,temp->arg2,temp->res);
        temp = temp->link;
        pos++;
    }
}
```

IV. Code Optimization

A character array as well as an array of elements of a quadruple structure is present to store the final optimized code. The character array is used to store each instruction completely so as to write it into a text file later. The quadruples structure is used to store the representation of each three-address code instruction so as to generate assembly code later.

There are 2 separate structures used for constant and copy propagation. Any time a variable has been assigned a constant value or been assigned another single variable, it is added to these structures so that in any further instruction when these variables are used they can be replaced with the value in the structure.

There is another structure which helps to implement dead code elimination. It eliminates unnecessary assignment statements. Every variable is stored in this structure, along with information about the last time it was used or assigned a value. In case it is assigned a new value without its old value being used, the old assignment instruction is eliminated.

```
char optim_code[1000][100];
int oc_pos = 0;

typedef struct copy_prop
{
    char var[20];
    char var_being_assign[20];
    struct copy_prop *link;
} cyp;
cyp *headcyp = NULL;

typedef struct cons_prop
{
    char var[20];
    char val_being_assign[20];
    struct cons_prop *link;
} csp;
csp *headcsp = NULL;
```

```
typedef struct vars_used
{
    char var[20];
    int line_assigned_start;
    int line_assigned_end;
    int has_been_used;
    int loop_no;
    struct vars_used *link;
} vu;
vu *headvu = NULL;

int line_st = 0;
int line_end = 0;
int no_of_arit_stat = 0;

typedef struct final_optim_code
{
    enum
    {
        assign, arit, branch_label,
        rel_exp, GOTO, if_goto,
        not, print
    } tag;

    char op[20];
    char arg1[20];
    char arg2[20];
    char res[20];
} foc;
foc optim_final[200];
int of_count = 0;
foc optim_final_assembly[200];
int ofa_count = 0;
```

Below are the functions used to obtain optimization. These functions are called in the functions defined to create the regular ICG to make further modifications and optimize the ICG.

```
//Optimization
void add_to_copy(char var[20], char val_being_assign[20]);
void add_to_cons(char var[20], char val_being_assign[20]);
int check_if_copy_or_cons_prop(char var[20], char return_val[20]);
int cons_or_var(char val[20]);

void remove_from_copy_lhs(char var[20]);
void remove_from_cons_lhs(char var[20]);
void remove_from_copy_rhs(char var[20]);
void remove_from_cons_rhs(char var[20]);

void remove_from_all(char var[20]);

void add_lines_to_arr(int st,int end);
void if_var_add(char var[20]);
void add_to_vars_list_assign(char var[20]);
void var_being_used(char var[20]);
void write_all_lines();

void write_optim_code(FILE *f_opt);

void create_final_optim_code();
```

The final optimized code is written into a text file.

```
void write_all_lines()
{
    vu *temp = headvu;
    while(temp)
    {
        if(temp->has_been_used == 0)
        {
            add_lines_to_arr(temp->line_assigned_start,temp->line_assigned_end);
        }
        temp=temp->link;
    }
}


void write_optim_code(FILE *f_opt)
{
    int i = 0;
    int k = 0;
    write_all_lines();
    while(i<oc_pos)
    {
        int flag = 0;
        for(int k=0;k<top_line;k++)
        {
            if(lines_to_remove[k] == i)
            {
                flag = 1;
                break;
            }
        }

        if(flag == 0)
        {
            fprintf(f_opt,"%s\n",optim_code[i]);

            optim_final_assembly[ofa_count] = optim_final[i];
            ofa_count++;
```

V. Error Handling

Syntax errors have been handled with grammar productions for missing semicolons or missing brackets, which place a call to yyerror.

```
semi                :   ';'
                    |   error { yyerror("Missing semicolon");}
                    ;

left_brac           :   '('
                    |   '{'
                    |   error { scope++; yyerror("Missing left bracket");}
                    ;

right_brac          :   ')'
                    |   '}'
                    |   error { scope_decrease = 1; yyerror("Missing right bracket");}
                    ;
```

Semantic errors have been handled with user-defined functions for multiple declarations of a variable or a variable not being declared but used. They are called as actions in the grammar productions, and based on their return value, if required yyerror is called.

```
declarator_list     :   value {
no_of_var_decl += 1;
int ch = check_multiple_assignments($1);
if(ch == 1){ yyerror("Multiple assignments");} this_var_is_being_declared = 1;} ',' declarator_list
                    |   value '=' arit_expression {
int ch = check_multiple_assignments($1);
if(undefined_var_in_arit == 1){ yyerror("Undefined Variable");}
else if(ch == 1){ yyerror("Multiple assignments");}
else { var_with_assign($1,$3);}
no_of_var_decl += 1; create_node_assign_init("="); undefined_var_in_arit = 0;}
```

The first function is called when a variable is being used. The variable is checked for whether it has been declared before or not. The second function is called when a variable is being declared to check if it has been declared before.

```
int check_if_var_exists(char *var)
{
    if(isdigit(var[0]))
    {
        return 1;
    }

    node *temp = head;
    while(temp)
    {
        if(strcmp(temp->id_name,var) == 0 && temp->scope == scope)
        {
            break;
        }
        temp = temp->link;
    }

    if(temp->has_been_declared == 0)
    {
        line_of_error = line_no;
        return 0;
    }
    return 1;
}

int check_multiple_assignments(char *var)
{
    if(multiple_declarations == 1)
    {
        multiple_declarations = 0;
        line_of_error = line_no;
        return 1;
    }
    return 0;
}
```

VI. Target Code Generation

Two structures have been used for registers. One is used to store all the available registers in a list and the other is used to store all the used registers and their values in a list.

```
// Assembly Code Generation
FILE *f_assembly;

typedef struct avail_reg_queue
{
    char reg[20];
    struct avail_reg_queue *link;
} arq;
arq *headarq = NULL;

typedef struct used_reg_list
{
    char reg[20];
    char var[20];
    int temp_var_st;
    struct used_reg_list *link;
} url;
url *headurl = NULL;

int reg_not_avail = 0;
```

Functions have been created to store registers in the corresponding list they belong to, as well as to shift them from one list to the other in case they are freed or being used. A function is also defined for each type of assembly instruction so as to generate the appropriate assembly language instruction. The entire assembly language is then written onto a text file.

```
// Assembly Code Generation
void create_reg_queue();
void check_reg_avail();
void free_all_temp_reg();
void add_reg_to_avail_reg(char reg[20]);

int assign_reg(char reg[20], char var[20]);
void get_reg(char reg[20], char var[20]);
void add_to_used_reg(char reg[20],char var[20]);
void get_first_used_reg(char reg[20]);

void add_arit_to_assembly(char res[20],char left_oper[20],char oper[20],char right_oper[20]);
void add_assign_to_assembly(char res[20],char val[20]);
void add_branch_label_to_assembly(char branch[20]);
void add_rel_to_assembly(char res[20],char left_oper[20],char oper[20],char right_oper[20], int i);
void add_goto_to_assembly(char res[20]);

void write_into_assembly();
```

VII. Common Tools

Every phase uses these common tools, which are file operations and string functions. Pre-defined functions are used from the string library such as strcmp and strcpy. Since every phase is stored in a text file, there is a need to use file functions such as fopen, fprintf and fclose.

VIII. Instructions to Run Program

- Instructions for running program to generate symbol table, tokens and AST:

  lex token_generator.l
  yacc -d grammar_validator.y
  gcc y.tab.c
  ./a.out

- Instructions for running program to generate ICG, optimized ICG and assembly code:

  lex icg.l
  yacc -d icg.y
  gcc y.tab.c
  ./a.out

RESULTS AND SHORTCOMINGS

Our project successfully compiles the input C++ program and generates all the required outputs, from generating tokens for each lexeme in the first phase of the compiler to generating target code in the last phase of the compiler.

While all possible cases were taken care of with respect to for loop, do-while loop and if conditional statement, we can further implement while loop or the if-else condition and improvise. User-defined data types such as arrays or structures have can further ne added to name a few.

SNAPSHOTS

- ***Input file without errors***

```
int main()
{
    int a,b=0;

    a=5;
    int c=b;
    for(int i=0;i<a;i=i+1)
    {
        b=1;
    }

    float z1=4.2,z2=4.8;


    do{
        b = b+2;
        c = 10;
        if(b == 5)
        {
            break;
        }
    }while(c<a);

    if(c<b)
    {
        if(a<b)
        {
            a = 10;
            cout<<a;
        }
    }
    int d = a/b+c;
```

- ***Input file with errors***

```
int main()
{
    int a,b=0;

    a=5
    int c=b;
    for(int i=0;i<a;i=i+1)
    {
        b=1;
    }

    float z1=4.2,z2=4.8;


    do{
        b = b+2;
        c = 10
        if(b == 5)
        {
            break;
        }
    }while(c<a;

    if(c<b
    {
        if(a<b)
        {
            a = 10;
            cout<<a;
        }
    }
    int d = a/b+c;
}
```

- ***Tokens Generated***

```
<KEYW, 'int', 1>
<KEYW, 'main', 1>
<SEP, '(', 1>
<SEP, ')', 1>
<SEP, '{', 2>
<KEYW, 'int', 3>
<ID, 'a', 1, 3>
<SEP, ',', 3>
<ID, 'b', 1, 3>
<ASSIGN, '=', 3>
<CONS, 'INT', 0, 3>
<SEP, ';', 3>
<ID, 'a', 1, 5>
<ASSIGN, '=', 5>
<CONS, 'INT', 5, 5>
<KEYW, 'int', 6>
<ID, 'c', 1, 6>
<ASSIGN, '=', 6>
<ID, 'b', 1, 6>
<SEP, ';', 6>
<KEYW, 'for', 7>
<SEP, '(', 7>
<KEYW, 'int', 7>
<ID, 'i', 2, 7>
<ASSIGN, '=', 7>
<CONS, 'INT', 0, 7>
<SEP, ';', 7>
<ID, 'i', 2, 7>
<RELOP, '<', 7>
<ID, 'a', 2, 7>
<SEP, ';', 7>
<ID, 'i', 2, 7>
<ASSIGN, '=', 7>
<ID, 'i', 2, 7>
<AROP, '+', 7>
<CONS, 'INT', 1, 7>
<SEP, ')', 7>
```

- ***Symbol table***

Identifier	Type	Value	Scope	Storage Req (bytes)	Line Number
a	int	10	1	4	3,5,32
b	int	3	1	4	3,6,32
c	int	10	1	4	6,32
i	int	1	2	4	7,7,7,7
a	int	10	2	4	7,22
b	int	3	2	4	9,16,16,24
z1	float	4.20	1	4	12
z2	float	4.80	1	4	12
c	int	10	2	4	17,22,24
b	int	3	3	4	18,26
a	int	10	3	4	26,28,29
d	int	0	1	4	32

- ***Errors generated and handled***

```
rishkj@Rishabhs-MacBook-Air Token_Generator and AST % ./a.out
At line no : 5
Error occured : syntax error
At line no : 5
Error occured : Missing semicolon
At line no : 17
Error occured : syntax error
At line no : 17
Error occured : Missing semicolon
At line no : 22
Error occured : syntax error
At line no : 22
Error occured : Missing right bracket
At line no : 24
Error occured : syntax error
At line no : 24
Error occured : Missing right bracket
Program accepted
```

- ***Abstract Syntax Tree***

```
<Main Tree>
    <Declaration>
        <Type> int
        <Declared Vars>
            <Var> a
            <Assign Exp>
                <Assign Exp oper> =
                <Var Getting Val>
                    <Var> b
                <Var Getting Val Closed>
                <Val Being Assigned>
                    <Int Constant> 0
                <Val Being Assigned Closed>
            <Assign Exp Closed>
        <Declared Vars Closed>
    <Declaration Closed>
    <Assign Exp>
        <Assign Exp oper> =
        <Var Getting Val>
            <Var> a
        <Var Getting Val Closed>
        <Val Being Assigned>
            <Int Constant> 5
        <Val Being Assigned Closed>
    <Assign Exp Closed>
    <Declaration>
        <Type> int
        <Declared Vars>
            <Assign Exp>
                <Assign Exp oper> =
                <Var Getting Val>
                    <Var> c
                <Var Getting Val Closed>
                <Val Being Assigned>
                    <Var> b
                <Val Being Assigned Closed>
            <Assign Exp Closed>
```

- ***ICG***

```
a = 0
b = 0
a = 5
c = b
i = 0
L0:
t0 = i < a
t1 = not t0
if t1 GOTO L1
b = 1
t2 = i + 1
i = t2
GOTO L0
L1:
z1 = 4.200000
z2 = 4.800000
L2:
t3 = b + 2
b = t3
c = 10
t4 = b == 5
t5 = not t4
if t5 GOTO L4
GOTO L3
L4:
t6 = c < a
if t6 GOTO L2
L3:
t7 = c < b
t8 = not t7
if t8 GOTO L5
t9 = a < b
t10 = not t9
if t10 GOTO L6
a = 10
print a
L6:
```

- ***Optimized ICG***

```
b = 0
a = 5
c = 0
i = 0
L0:
t0 = i < a
t1 = not t0
if t1 GOTO L1
b = 1
i = i + 1
GOTO L0
L1:
L2:
b = b + 2
c = 10
t4 = b == 5
t5 = not t4
if t5 GOTO L4
GOTO L3
L4:
t6 = c < a
if t6 GOTO L2
L3:
t7 = 10 < b
t8 = not t7
if t8 GOTO L5
t9 = 5 < b
t10 = not t9
if t10 GOTO L6
print 10
L6:
L5:
```

- ***Assembly Code***

```
MOV R0, #0
ST b, R0
MOV R1, #5
ST a, R1
MOV R2, #0
ST c, R2
MOV R3, #0
ST i, R3
L0: SUB R4, R3, R1
BGEZ R4, L1
MOV R0, #1
ST b, R0
MOV R5, #1
ADD R3, R3, R5
ST i, R3
BR L0
L1: L2: MOV R6, #2
ADD R0, R0, R6
ST b, R0
MOV R2, #10
ST c, R2
MOV R7, #5
SUB R8, R0, R7
BNE R8, L4
BR L3
L4: SUB R9, R2, R1
BLTZ R9, L2
L3: MOV R10, #10
SUB R11, R10, R0
BGEZ R11, L5
SUB R12, R7, R0
BGEZ R12, L6
L6: L5:
```

CONCLUSION

A mini-compiler for C++ using C has successfully been created. Our input C++ program is parsed through completely and all the phases of a compiler have been performed successfully. Tokens are generated, panic mode recovery is implemented for error recovery, an abstract syntax tree is created, intermediate code is generated and optimized and finally assembly language instructions are created. All of this has been done using lex and yacc.

FURTHER ENHANCEMENTS

Some further enhancements that could be made are:

- Constructs such as while loops, switch statements or if-else conditions could be added to our grammar, as well as creating user defined functions.
- Some more optimizations could have been implemented, such as constant folding.
- Can add implementation for user-defined data types.