Report On

"mini C++ compiler (switch case)"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Joshua Phillips	**PES2201800333**
S Deepashree	**PES2201800153**
Ketan Malempati	**PES2201800088**

Under the guidance of

Prof. Swati G

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

1. INTRODUCTION

We have built a self-compiling mini-compiler for C++.

C++ is a general purpose programming language and widely used now a days for competitive programming. It has imperative, object-oriented and generic programming features.
The main constructs that we have focused on while building this compiler is ‘switch’. The compiler also identifies arithmetic, boolean and logical operations.
The expected outcome of this project is to generate a Symbol Table, an Abstract Syntax Tree and Intermediate Three-Address code along with optimization.

2. ARCHITECTURE

The mini compiler handles most cases in the C++09

Features Of The Lexer.

1. Identifies and removes comments
2. Identifies various operators in the language
3. Checks for validity of the identifiers
4. Identifying numeric constants and std::string type
5. Ignores white-spaces
6. Identifies scope of variables

Syntax is handled by YACC where grammar rules are specified for the entire language.
Semantics are handled using semantic rules for type checking while performing operations, to ensure operations are valid.

3. LITERATURE SURVEY

1. Lex & Yacc, O’Reilly, John R. Levine, Tony Mason, Doug Brown

4. CONTEXT FREE GRAMMAR

```
start: T_INT T_MAIN T_OPEN T_CLOSE comp_stat  {$$ = make_leaf($1); $
$=make_node("Main",$2,$5);
```

```
        printf("\n\n");print_sym_tab(); YYACCEPT;}
    ;

comp_stat: T_OPBRACE SCOPE stat T_CLBRACE {$
$=$3;scope[scope_ind++]=0;}
                ;

SCOPE: {scope_ctr++;scope[scope_ind++]=scope_ctr;}
        ;
stat:E T_SC stat             {$$=make_node("Statement",$1,$3);}
   |assign_expr stat       {$$=make_node("Statement",$1,$2);}
   |comp_stat stat         {$$=make_node("Statement",$1,$2);}
   |select_stat stat       {$$=make_node("Statement",$1,$2);}
   |decl stat                       {$$=make_node("Statement",$1,$2);}
   |jump_stat stat         {$$=make_node("Statement",$1,$2);}
   |                          {$$=make_leaf("  ");}
   | error T_SC
   ;


ST  : T_SWITCH T_OPEN T_ID T_CLOSE T_OPBRACE  {scope_ctr+
+;scope[scope_ind++]=scope_ctr;}  B T_CLBRACE {scope[scope_ind+
+]=0;$$=make_leaf($1);
                                                            $$=make_node("Switch",
(AST_node*)$5,$7);if(!look_up_sym_tab($3)){printf("Undeclared variable
%s\n", $3); YYERROR;}}
   ;

B   : C        {$$=$1;}
   | C D       {$$=make_node("Cases",$1,$2);}
   | C B       {$$=make_node("Cases",$1,$2);}
   ;

C   : T_CASE T_NUM T_COLON stat     {$$=make_node("Case",
(AST_node*)$2,$4);}
   ;

D   : T_DEFAULT T_COLON stat      {$1=make_leaf(" "); $
$=make_node("Default",$1,$3);}
   ;

select_stat: ST   {$$=$1;}
                 ;
```

```
jump_stat:T_CONTINUE T_SC            {$$=make_leaf("Continue");}
                  |T_BREAK T_SC      {$$=make_leaf("Break");}
                  |T_RETURN E T_SC    {$1=make_leaf("Return");$$ =
make_node("Statement",$1,$2);}
                  ;

decl:Type Varlist T_SC    {$1=(char*)make_leaf($1); $$=make_leaf($2); $
$=make_node("Variable_Declaration",(AST_node*)$1,(AST_node*)$2); }
    |Type assign_expr1   {$1=(char*)make_leaf($1); $
$=make_node("Variable_Declaration",(AST_node*)$1,$2);}
    ;

Type:T_INT          {$$ = $1;strcpy(typ,$1);}
        |T_FLOAT      {$$ = $1;strcpy(typ,$1);}
        |T_DOUBLE {$$ = $1;strcpy(typ,$1);}
        |T_CHAR              {$$ = $1;strcpy(typ,$1);}
        ;

Varlist:Varlist T_COMMA T_ID    {$3=(char*)make_leaf($3);$
$=(char*)make_node("Variable List",(AST_node*)$1,(AST_node*)
$3);if(look_up_sym_tab_dec($3,scope[scope_ind-1]))
{ yyerror("Redeclaration\n");  YYERROR; }
                                                          if(scope[scope_ind-1]>0)
{update_sym_tab($<var_type>0,$3,yylineno,scope[scope_ind-
1]);}else{int
scop=get_scope();update_sym_tab($<var_type>0,$3,yylineno,scop);}}
          |T_ID                         {$$=(char*)make_leaf($1);
if(look_up_sym_tab_dec($1,scope[scope_ind-1])){ yyerror("Redeclaration\
n");  YYERROR; }
                                            if(scope[scope_ind-1]>0)
{update_sym_tab($<var_type>0,$1,yylineno,scope[scope_ind-
1]);}else{int
scop=get_scope();update_sym_tab($<var_type>0,$1,yylineno,scop);}}


assign_expr:T_ID T_ASSIGN E T_COMMA assign_expr
{$1=(char*)make_leaf($1); $$=make_for_node($2,(AST_node*)$1,
(AST_node*)$3,make_leaf(","),$5); if(!look_up_sym_tab($1))
{printf("Undeclared variable %s\n", $1); YYERROR;}}
                  |T_ID T_ASSIGN E T_SC           {$1=(char*)make_leaf($1); $
$=make_node($2,(AST_node*)$1,(AST_node*)$3); if(!
look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1); YYERROR;}
                                                                                      }
```

```
                    ;

assign_expr1:T_ID T_ASSIGN E T_COMMA assign_expr1
{$1=(char*)make_leaf($1);$$=make_for_node($2,(AST_node*)$1,
(AST_node*)$3,make_leaf(","),$5);
if(look_up_sym_tab_dec($1,scope[scope_ind-1])){ yyerror("Redeclaration\
n");  YYERROR; }
if(scope[scope_ind-1]>0){update_sym_tab(typ,
$1,yylineno,scope[scope_ind-1]);}else{int
scop=get_scope();update_sym_tab(typ,$1,yylineno,scop);}}
                    |T_ID T_ASSIGN E T_SC          {$1=(char*)make_leaf($1);$
$=make_node($2,(AST_node*)$1,(AST_node*)$3);
if(look_up_sym_tab_dec($1,scope[scope_ind-1])){ yyerror("Redeclaration\
n");  YYERROR; } if(scope[scope_ind-1]>0){update_sym_tab(typ,
$1,yylineno,scope[scope_ind-1]);}else{int
scop=get_scope();update_sym_tab(typ,$1,yylineno,scop);} }
                    ;

E:E T_PLUS T    {$$=make_node($2,$1,$3); }
 |E T_MINUS T    {$$=make_node($2,$1,$3);}
 |T           {$$=$1;}
 ;

T:T T_MULT F    {$$=make_node($2,$1,$3);}
 |T T_DIV F     {$$=make_node($2,$1,$3);}
 |F           {$$=$1;}
 ;

F:T_ID  {$$=make_leaf($1); if(!look_up_sym_tab($1)){printf("Undeclared
Variable %s\n", $1); YYERROR;} }
 |T_NUM           {$$=make_leaf($1);}
 |T_OPEN E T_CLOSE   {$$=$2;}
 |unary_expr       {$$=$1;}
 |s_operation      {$$=$1;}
 ;

s_operation: T_ID s_op T_ID  {$1=(char*)make_leaf($1);
$3=(char*)make_leaf($3); $$=make_node($2,(AST_node*)$1,(AST_node*)
$3); if(!look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}if(!look_up_sym_tab($3)){printf("Undeclared variable %s\n",
$3); YYERROR;}}
                             | T_ID s_op T_NUM {$1=(char*)make_leaf($1);
$3=(char*)make_leaf($3); $$=make_node($2,(AST_node*)$1,(AST_node*)
```

```
$3); if(!look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}}
                              | T_ID s_op T_OPEN E T_CLOSE
{$1=(char*)make_leaf($1); $$=make_node($2,(AST_node*)$1,
(AST_node*)$4);}
                              ;

s_op:T_SPLUS   {$$=$1;}
         |T_SMINUS   {$$=$1;}
         |T_SMULT   {$$=$1;}
         |T_SDIV    {$$=$1;}
         ;

unary_expr:T_INC T_ID {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2); if(!
look_up_sym_tab($2)){printf("Undeclared variable %s\n", $2);
YYERROR;}}
                    |T_ID T_INC {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2); if(!
look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}}
                    |T_DEC T_ID {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2); if(!
look_up_sym_tab($2)){printf("Undeclared variable %s\n", $2);
YYERROR;}}
                    |T_ID T_DEC {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2); if(!
look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}}
                    | T_MINUS T_ID {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2); if(!
look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}}
                    | T_MINUS T_NUM   {$$=make_leaf($1); $$=make_leaf($2);$
$=make_node("temp",(AST_node*)$1,(AST_node*)$2);}
                    ;
```

5. DESIGN STRATEGIES

• SYMBOL TABLE

The Symbol Table is used for storing variables declared and their attributes, along with details about function calls. The Symbol table stores the size of a variable, it's scope and also the line numbers where the particular variable is used.
We used hash tables to implement the symbol table. The variable names were hashed and allocated a fixed amount of space in the table. Every new scope has it's own symbol table, which would be destroyed once the scope would end. All symbol tables are connected using an n-ary tree. So basically, we have implemented an n-ary tree of hash tables. Starting with the root node which has the global symbol table.

Upon encountering a new scope, a new hash table is created as a child node of the root node. If within the same scope another new scope is encountered a new child node of the current scope node is created, and so on. Sibling nodes (i.e child nodes on the same level) have different hash tables and hence cannot access each other's data.
But a child node can access data from it's parent hash table.

• INTERMEDIATE CODE GENERATION

To convert our given C language into intermediate code, we have used the SDT scheme and made use of marker non-terminals in place of action records.
Label generation:

We have made use of stack whose elements are of type records.
Each entry to the stack is a hash table for which we have used maps.
Each of marker non-terminal will push a map record to the stack which contains

• CODE OPTIMIZATION

We have implemented Dead Code elimination for the ICG generated. Dead codes are pieces of code that contain temporaries that are not used further or anywhere else in the generated ICG. We keep track of all the useful temporaries and hence when we encounter a line which uses a non-useful temporary, we can eliminate that line of code from the ICG.

• ERROR HANDLING

We are handling syntax errors, which are generated during parsing. We are also handling re-declaration of variables in the same scope, and are showing appropriate error messages. We stop parsing the input on encountering these errors and display the line number of the errors, intending the user to resolve the issue. Also handling type mismatch errors.

6. IMPLEMENTATION DETAILS.

- **SYMBOL TABLE**

<u>Data Structure Implementation</u>

Structure to point to reference of each variable occurrence in program

```
struct node
{
	char token[100];
	char attr[100];
	struct node *next;
};
```

<u>Functions</u>

Functions that have been written for the symbol table

```
struct node * createNode(char *token, char *attr)
{
	struct node *newnode;
	newnode = (struct node *) malloc(sizeof(struct node));
	strcpy(newnode->token, token);
	strcpy(newnode->attr, attr);
	newnode->next = NULL;
	return newnode;
}

int hashIndex(char *token)
{
	int hi=0;
	int l,i;
	for(i=0;token[i]!='\0';i++)
	{
		hi = hi + (int)token[i];
	}
	hi = hi%eleCount;
	return hi;
}
```

```
void insertToHash(char *token, char *attr)
{
	int flag=0;
	int hi;
	hi = hashIndex(token);
	struct node *newnode = createNode(token, attr);
	/* head of list for the bucket with index "hashIndex" */
	if (hashTable[hi].head==NULL)
	{
		 hashTable[hi].head = newnode;
		hashTable[hi].count = 1;
		return;
	}
	struct node *myNode;
	myNode = hashTable[hi].head;
	while (myNode != NULL)
	{
	if (strcmp(myNode->token, token)==0)
	{
			flag = 1;
			break;
	}
	myNode = myNode->next;
	}
	if(!flag)
	{
		//adding new node to the list
		newnode->next = (hashTable[hi].head);
		//update the head of the list and no of nodes in the current
bucket
		hashTable[hi].head = newnode;
		hashTable[hi].count++;
	}
	return;
}
```

<u>Functions Implemented</u>

All the functions that have been implemented for Symbol Table

```
struct node * createNode(char *token, char *attr)
int hashIndex(char *token)
void insertToHash(char *token, char *attr)
```

- **INTERMEDIATE CODE GENERATION**

Data Structure Implementation
Following is the type of non-terminals used in our grammar.

```
struct exprType{
        char *addr;
        char *code;

};
```

Functions
Functions that have been written for the Intermediate Code Generator

```
//Function to generate new temporary variables
char * newTemp(){
        char *newTemp = (char *)malloc(20);
        strcpy(newTemp,"t");
        num_to_concatinate[0]=0;
        snprintf(num_to_concatinate, 10,"%d",n);
        strcat(newTemp,num_to_concatinate);
        n++;
        return newTemp;
}

//Function to generate new labels
char * newLabel(){
        char *newLabel = (char *)malloc(20);
        strcpy(newLabel,"L");
        snprintf(num_to_concatinate_l, 10,"%d",nl);
        strcat(newLabel,num_to_concatinate_l);
        nl++;
        return newLabel;
}

//Function to replace a substring str with another substring label in the
original string s1
void replace(char* s1,char* str, char* label)
{
        char* check = strstr (s1,str);
                        while(check!=NULL){
                        strncpy (check,label,strlen(label));
                        strncpy (check+strlen(label),"   ",(4-strlen(label)));
                        check = strstr (s1,str);}
```

```
}
```

Functions Implemented

All the functions that have been implemented for Intermediate Code Generator

```
char * newTemp()
char * newLabel()
void replace(char* s1,char* str, char* label)
```

- **CODE OPTIMIZATION**

Data Structure Implementation

Following is the type of non-terminals used in our grammar.

```
union {
        int ival;
        float fval;
        char *sval;
        struct exprType *EXPRTYPE;
}
```

- **ERROR HANDLING**

Function to handle syntax errors, gets called automatically by yacc on encountering syntax errors

```
void yyerror(const char* msg)
{
   printf("%s", msg);
}
```

For re-declaration check we used a flag variable declare=0, which would be set and unset accordingly in the same scope. If it was previously set and is set again in the same scope, we can catch the error and display appropriate error messages.

- **RUNNING CODE**
  - Running Lexer

```
lex lexer.l
gcc lex.yy.c
./a.out testcase1.cpp
```

- Running Parser

```
flex switch.l
yacc -d switch.y
gcc y.tab.c lex.yy.c -w
./a.out < testcase1.cpp
```

- Running Intermediate Code Generator

```
flex icg_ket.y
yacc -d icg_ket.y
gcc y.tab.c -ll -ly -w
./a.out > output.txt
```

- Running Code Optimizer
  - For Dead Code Elimination
  - Constant Folding

```
python codeopt.py
```

7. RESULTS

We were able to successfully generate the different representations of input code written in C++ along with performing simple machine independent optimization on intermediate code which was generated.

<u>Shortcomings</u>

1. Error handling could have been done better by adding more rules.
2. There could've been more ways for us to implement Code Optimization techniques

8. SNAPSHOTS

- **INPUT**

```
#include <iostream>

int main()
{
    char oper = '+';
    float num1 = 3, num2 = 5;
    switch (oper) {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1 + num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1 - num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1 * num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1 / num2
            break;
        default:
            // operator is doesn't match any case constant (+, -, *, /)
            cout << "Error! The operator is not correct";
            break;
    }
    return 0;
}
```

- **SYMBOL TABLE**

```
        ******** SYMBOL TABLE ********
-----------------------------------------------------------------
SNo       |       Token          |       Token Type
-----------------------------------------------------------------
1               (                       SPECIAL SYMBOL
2               )                       SPECIAL SYMBOL
3               0                       INTEGER CONSTANT
4               1                       INTEGER CONSTANT
5               3                       INTEGER CONSTANT
6               5                       INTEGER CONSTANT
7               7                       INTEGER CONSTANT
8               ;                       SPECIAL SYMBOL
9               =                       OPERATOR
10              56                      INTEGER CONSTANT
11              x                       IDENTIFIER
12              y                       IDENTIFIER
13              z                       IDENTIFIER
14              {                       SPECIAL SYMBOL
15              }                       SPECIAL SYMBOL
16              209                     INTEGER CONSTANT
17              case                    KEYWORD
18              break                   KEYWORD
19              switch                  KEYWORD
20              default                 KEYWORD
-----------------------------------------------------------------
```

- **TOKENS GENERATED FROM LEXER**

```
x              IDENTIFIER          Line 1
=              OPERATOR            Line 1
3              INTEGER CONSTANT    Line 1
;              SPECIAL SYMBOL      Line 1
y              IDENTIFIER          Line 2
=              OPERATOR            Line 2
5              INTEGER CONSTANT    Line 2
;              SPECIAL SYMBOL      Line 2
z              IDENTIFIER          Line 3
=              OPERATOR            Line 3
0              INTEGER CONSTANT    Line 3
;              SPECIAL SYMBOL      Line 3
switch         KEYWORD             Line 5
(              SPECIAL SYMBOL      Line 5
x              IDENTIFIER          Line 5
)              SPECIAL SYMBOL      Line 5
{              SPECIAL SYMBOL      Line 5
case           KEYWORD             Line 6
0              INTEGER CONSTANT    Line 6
:z             IDENTIFIER          Line 7
=              OPERATOR            Line 7
3              INTEGER CONSTANT    Line 7
;              SPECIAL SYMBOL      Line 7
break          KEYWORD             Line 8
case           KEYWORD             Line 9
1              INTEGER CONSTANT    Line 9
:z             IDENTIFIER          Line 10
=              OPERATOR            Line 10
56             INTEGER CONSTANT    Line 10
;              SPECIAL SYMBOL      Line 10
break          KEYWORD             Line 11
;              SPECIAL SYMBOL      Line 11
case           KEYWORD             Line 12
3              INTEGER CONSTANT    Line 12
:z             IDENTIFIER          Line 13
=              OPERATOR            Line 13
209            INTEGER CONSTANT    Line 13
;              SPECIAL SYMBOL      Line 13
break          KEYWORD             Line 14
;              SPECIAL SYMBOL      Line 14
case           KEYWORD             Line 15
7              INTEGER CONSTANT    Line 15
```

- **PARSER OUTPUT**

```
Parsing the following Input:

int main() {
    char oper;
    float num1, num2;
    cout << "EnterUndeclared Variable cout
 an operator (+, -, *, /): ";
    cin >> oper;
    cout << "Enter two numbers: " << endl;
    cin >> num1 >> num2;

    switch (oper) {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1 + num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1 - num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1 * num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1 / num2;
            break;
        default:
            (+, -, *, /)
            cout << "Error! The operator is not correct";
            break;
    }



Symbol Table:
Token: oper, Type: char, Size: 1, Line Number: 2, Scope: 1

Token: num1, Type: float, Size: 8, Line Number: 3, Scope: 1

Token: num2, Type: float, Size: 8, Line Number: 3, Scope: 1

Success
```

- **INTERMEDIATE CODE GENERATOR**

Input	Output

switch(a)
{
case 0: {a=b+c;}
case 1: {p=q+r;}
default:
i=0;
}
	---------- FINAL THREE ADDRESS CODE ----------
	
	if(a =0) goto L3
	goto L4
	L3 : t1=b+c
	a=t1
	goto L5
	L4 :
	if(a =1) goto L1
	goto L2
	L1 : t3=q+r
	p=t3
	goto L5
	L2 : i=0
	L5 : y=0
	L6 : END OF THREE ADDRESS CODE !!!!!

```
----------  FINAL THREE ADDRESS CODE ----------

if(a   =0) goto L5
goto L6
L5 : t1=b+c
a=t1
goto L7
L6 :
if(a   =1) goto L3
goto L4
L3 : t3=q+r
t4=t3-y
p=t4
goto L7
L4 :
if(a   =2) goto L1
goto L2
L1 : t6=h*u
t7=t6-d
t8=t7+e
g=t8
goto L7
L2 : i=0
L7 : END OF THREE ADDRESS CODE !!!!!

python a.py
*********Quadruple *************
operator operand1       operand2        result
=               0               NULL            a
+               b               c               t1
=               1               NULL            a
+               q               r               t3
=               2               NULL            a
*               h               u               t6
=               0               NULL            i
```

- **CODE OPTIMIZER**

```
Quadruple form after Constant Folding
-------------------------------------
('=', '0', 'NULL', 'a')
('+', 'b', 'c', 't1')
('=', '1', 'NULL', 'a')
('+', 'q', 'r', 't3')
('=', '2', 'NULL', 'a')
('*', 'h', 'u', 't6')
('=', '0', 'NULL', 'i')


Constant folded expression -
--------------------
('a', '=', '0')
('t1', '=', 'b', '+', 'c')
('a', '=', '1')
('t3', '=', 'q', '+', 'r')
('a', '=', '2')
('t6', '=', 'h', '*', 'u')
('i', '=', '0')


After dead code elimination -
------------------------------
('t1', '=', 'b', '+', 'c')
('t3', '=', 'q', '+', 'r')
('t6', '=', 'h', '*', 'u')
```

9. CONCLUSIONS

By doing this project, we have gained a better insight into the phases of the compiler. YACC provided us with a better knowledge about bottom-up parser and while performing the different phases of the compiler, our code efficiency in writing code and dealing with complex data structures has significantly improved.

10. FURTHER ENHANCEMENTS

Would like to include more ways of Code Optimizer

11. REFERENCES/BIBLIOGRAPHY

Lex & Yacc, O'Reilly, John R Levine, Tony Mason, Doug Brown

Our Code can be found on https://github.com/Joshua-Phillips1999/CompilerDesign