Report on

"Developing a mini-compiler for the C language"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Manish Shetty	**01FB16ECS192**
Midhush M	**01FB16ECS208**
Nishant R S	**01FB16ECS236**

Under the guidance of

Ms. Sangeeta V I
Assistant Professor
PES University, Bengaluru

January – May 2019

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

C was the language chosen as the basis of this mini compiler. Simple constructs from the language were implemented. The frontend of the compiler including the 4 phases ie:

1. Symbol table generation
2. Abstract Syntax tree construction
3. Intermediate Code generation
4. Intermediate Code Optimization

was implemented using flex and bison. The language the compiler is designed for is C. The properties we have implemented includes the basic structure of C which includes initializing variables, creating main and other functions, preprocessor directives limited to including <stdio.h> and so on.

The specific constructs chosen for implementation other than all basic structural elements of the language were :

1. if-else as selection statement
2. do-while as looping statement

2. ARCHITECTURE OF LANGUAGE

Compiler for the following constructs:

- `do while` loop
- `if else` statements
- `int`, `char` data types
- `basic function` declaration and definition
- arithmetic , logical and relational operators

* Removal of comments has been taken care of too *

3. REFERENCES

Lex Yacc and its internal working

https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc1

Building a mini-compiler - tutorial

https://www.tutorialspoint.com/compiler_design/index.htm

Expression evaluation using Abstract Syntax Tree

https://mariusbancila.ro/blog/2009/02/03/evaluating-expressions-part-1

4.CONTEXT-FREE GRAMMAR

```
%%

start_state
        : HEADER start_state
        | translation_unit
        ;

translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;


external_declaration
        : function_definition
        | declaration
        ;

function_definition
        : type_specifier declarator '(' params ')' compound_statement
        ;

params
        : param_decl ',' params
        | param_decl
        ;

param_decl
        : type_specifier declarator
        | type_specifier
        ;

function_call
        : declarator '(' varList ')'
        ;

varList
        : varList ',' declarator
        | declarator
        ;

declarator
        : IDENTIFIER
        ;

declaration
        : type_specifier init_declarator_list ';'
        | type_specifier ';'
        ;

init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;
```

```
init_declarator
        : declarator '=' primary_expression
        | declarator '=' simple_expression
        | declarator '=' function_call
        | declarator
        ;

type_specifier
        : VOID
        | CHAR
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | UNSIGNED INT
        | UNSIGNED SHORT INT
        | UNSIGNED LONG INT
        | UNSIGNED LONG LONG INT
        | SIGNED INT
        | SIGNED SHORT INT
        | SIGNED LONG INT
        | SIGNED LONG LONG INT
        ;

primary_expression
        : declarator
        | CONSTANT
        ;

simple_expression
        : simple_expression OR and_expression
        | and_expression
        ;

and_expression
        : and_expression AND unary_rel_expression
        | unary_rel_expression
        ;

unary_rel_expression
        : NOT factor
        | rel_expression
        ;

rel_expression
        : sum_expression
        | sum_expression RELOP sum_expression
        ;

sum_expression
        : sum_expression sumop term
        | term
        ;
```

```
sumop
        : '+'
        | '-'
        ;

logop
        : OR
        | AND
        ;

term
        : term mulop factor
        | factor
        ;

mulop
        : '*'
        | '/'
        ;

factor
        : primary_expression
        | '(' simple_expression ')'
        ;


compound_statement
        : '{' '}'
        | '{' block_scope_list '}'
        ;
```

```
block_scope_list
        : block_item
        | block_item block_scope_list
        ;

block_item
        : declaration
        | statement
        ;

statement
        : expression_statement
        | compound_statement
        | conditional_statement
        | iteration_statement
        | break_statement
        | continue_statement
        | return_statement
        | statement ';' statement
        | function_call
        ;
```

```
expression_statement
        : expression ';'
        | ';'
        ;

expression
        : declarator '=' expression
        | simple_expression
        | declarator '=' function_call
        ;

conditional_statement
        : IF '(' condition ')' compound_statement %prec REDUCE
        | IF '(' condition ')' compound_statement ELSE  compound_statement
        ;

condition
        : expression logop expression
        | expression
        ;

iteration_statement
        : DO compound_statement  WHILE '(' condition ')' ';'
        ;

break_statement
        : BREAK ';'
        ;

continue_statement
        : CONTINUE ';'
        ;

return_statement
        : RETURN ';'
        | RETURN simple_expression ';'
        ;

%%
```

5. DESIGN STRATEGY

● **Symbol table creation**- The symbol table was implemented in a linear array structure that contains the identifier and its value. Function scope has been implemented as well using a counter to increment and decrement scope number.

● **Abstract Syntax Tree**- This tree is constructed as the input is parsed using semantic rules. Each node of this tree contains a pointer to left, a pointer to right and a member for a string.

● **Intermediate Code Generation**- Intermediate code was generated that makes use of temporary variables and labels. Also all if-else statements were optimized to ifFalse statements to reduce the number of goto statements to the true and false cases (This is an additional optimization provided that is inbuilt with the ICG itself).

● **Code Optimization-** Constant folding and Constant propagation were implemented as part of machine independent code optimization. To do so the ICG was first converted to quadruple form.

NOTE : A specific converter was written in C++ to convert a given ICG to a object of Quadruple Class type , which has methods that can be used to optimize on the quadruples

Constant Folding In the Quadruple table , a statement was flagged to be a constant expression if it has only numeric arguments. Such expressions were evaluated at the moment and stored back in the quadruple in the correct syntactic form. As soon as some constant folding took place or a variable that was declared with a constant value was encountered it was moved to the next phase of optimization.

Constant Propagation On encountering a variable that evaluated to a constant it was propagated to all statements that occur after it until it reappears in the table.

In the above fashion we have mode the optimization phase quicker by combining the folding and propagations and thus the code boils down to its final state faster.

● **Error handling-** On encountering any errors in the program , the parse fails and the line number at which the error occurred is printed

6. IMPLEMENTATION DETAILS

The various components of the project were implemented separately as debugging became easier. Also by doing so , the individual components could be worked on without affecting the others in a modular fashion. Each individual component has been illustrated below as their respective data structures used and the functions that work on the.

- **Symbol Table & Scope Table**
  - ***a. Data Structures***

```
struct node
{
    char type[100];
    char symbol[100];
    int value; // should be made union for different types
    int first_line_num;
    int last_line_num;
    int storage_req; // should accomodate for different stuff
    int scope, parentScope, funcScope;
    int valid; // 1 - valid, 0 - invalid
    long startAddress; // For function it will be the Line number in three address code
  int params;   // Number of parameters in a function
};

struct scopeTable{
  int scopeNum;
  int num;
  struct node symArr[MAX];
  struct scopeTable *parent;
  struct scopeTable *children[100];
  int numChild;
};
```

  - ***b. Functions***

```
int addSym(struct scopeTable *head, char* givenSymbol,
          int lineno, char* givenType, int givenValue,
          int str_req, int scope);

int setVal(struct scopeTable *head, char *symbol, int lineno, int givenValue, int str_req);

int superAdd(struct scopeTable *head, char* givenSymbol,
            int add, int lineno, char* givenType,
            int givenValue, int str_req, int scope);
int findLen(struct scopeTable *head, char *givenId); // To find the length of an array
struct scopeTable* addScope(struct scopeTable *head, int scopeNumber);
struct scopeTable* leaveScope(struct scopeTable *head);
struct node getVal(struct scopeTable *head, char *symbol, int *succ, int lineno);
int printsymtab(struct scopeTable *head);
```

- **Abstract Syntax Tree-** Implemented in ast.y

```
int ex (nodeType *p, int flag);
// prototype to create a node for an operation
nodeType *opr(int oper, int nops, ...);
// prototype to create a node for an identifier
nodeType *id(char *identifier);
// prototype to create a node for a constat
nodeType *con(char *value);

typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct
{
        char value[100]; /* value of constant */
} conNodeType;

/* identifiers */
typedef struct
{
        char name[100]; /* subscript to sym array */
} idNodeType;

/* operators */
typedef struct
{
        int oper; /* operator */
        int nops; /* number of operands */
        struct nodeTypeTag *op[1]; /* operands, extended at runtime */
} oprNodeType;

typedef struct nodeTypeTag
{
        nodeEnum type; /* type of node */
        union
        {
                conNodeType con; /* constants */
                idNodeType id; /* identifiers */
                oprNodeType opr; /* operators */
        };
} nodeType;
```

- **Intermediate Code Generation**- Implemented in icg.y The given code was converted into 3 address code using a code on the fly approach.

a. Data structures

```
//stack of variables
char st[1000][10];
int top=0;
int i=0;

//temporary variable for t0,t1 ...
char temp[2]="t";

//array of labels used
int label[200];
int lnum=0; //label number
int ltop=0; //keep track of label at top

//copy to top of stack
void push(){ strcpy(st[++top],yytext);}
```

b. Functions Used

I. ICG for basic statements and functions

```
void codegen_assign()
{
  fprintf(f1,"%s\t=\t%s\n",st[top-1],st[top]);
  top-=2;
}

void codegen_func_def()
{
  fprintf(f1,"func\tbegin\t%s\n",st[top]);
  top-=1;
}

//a ( </>/+/* ...) b
void codegen_logical()
{
  sprintf(temp,"t%d",i);
  fprintf(f1,"%s\t=\t%s\t%s\t%s\n",temp,st[top-2],st[top-1],st[top]);
  top-=2;
  strcpy(st[top],temp);
  i++;
}
```

```
  void codegen_param(int n)
{
  int l = n;
  while(l--)
  {
    fprintf(f1,"param\t%s\n", st[top-l]);
  }
  top=top-n;
}

void codegen_function_name(int n,int hasReturnType)
{
  if(ISFUNCCALL > 0) {
    if(hasReturnType) {
      fprintf(f1,"%s\t=\tcall %s %d\n",st[top-1],st[top],n);
      top-=2;
    } else {
      fprintf(f1,"call %s %d\n",st[top],n);
      top-=1;
    }
  }
}
```

II. ICG for selection statement

```
void codegen_conditional_if()
{
  lnum++;
  label[ltop] = lnum;
  ltop++;
  fprintf(f1, "ifFalse %s goto L%d\n", temp, lnum);
  top-=1;
}
void codegen_conditional_else()
{
  lnum++;
  fprintf(f1, "goto L%d\n", lnum);
  fprintf(f1, "L%d:\n", label[ltop-1]);
  label[ltop] = lnum;
}
void codegen_conditional_end()
{
  fprintf(f1, "L%d:\n", label[ltop-1]);
  ltop-=1;
}
```

III. ICG for iteration

```
void codegen_iterational_begin()
{
  lnum++;
  label[ltop] = lnum;
  ltop++;
  fprintf(f1, "L%d:\n", label[ltop-1]);
}
void codegen_iterational_end()
{
  fprintf(f1, "if %s goto L%d\n", temp, label[ltop-1]);
  ltop-=1;
}
```

- **Code Optimization-** implemented in icg_to_quadruple.cpp . TAC was first converted to a quadruple table. In the Quadruple table , a statement was flagged to be a constant expression if it has only numeric arguments. Such expressions were evaluated at the moment and stored back in the quadruple in the correct syntactic form. As soon as some constant folding took place or a variable that was declared with a constant value was encountered it was moved to the next instructions and checked for propagation.

a. Data structures

```
class quadruple
{

public:

  vector< vector<string> > table;

  void show_table()
  {
    cout << "operator" << "      " << "arg1" << "        " << "arg2" <<"          " << "result" << endl;
    for (int i = 0; i < table.size(); i++)
    {
        for (int j = 0; j < table[i].size(); j++)
        {
            if(table[i][j]=="ifFalse") cout << table[i][j] << "        ";
            else if(table[i][j]=="param" || table[i][j]=="call" ) cout << table[i][j] << "        ";
            else cout << table[i][j] << "             ";
        }
        cout << endl;
    }
    cout << endl;
  }


};
```

b. Functions Used

```
// creates quad from icg
quadruple create_quadruples(string filename);
// evaluates const expression
int eval(string arg1 , string op , string arg2);
// constant propagation
quadruple constant_propogate( quadruple q , int i);
// constant folding
quadruple constant_optimize(quadruple q);
```

7. RESULTS

Sample Code 1

```c
#include <stdio.h>

int func(int c,int d) {
  int a = 0; // this is a comment
  return 0;
}
```

After removal of comments

```
1#include <stdio.h>
2
3 int func ( int c , int d ) {
4    int a = 0 ;
5    return 0 ;
6 }
```

ICG

```
func    begin   func
param   c
param   d
a   =   0
```

Quadruple Table and optimization

```
QUADRUPLE TABLE
operator      arg1       arg2          result
param         c
param         d
=              0                          a

AFTER CONSTANT FOLDING AND PROPAGATION
operator      arg1       arg2          result
param         c
param         d
=              0                          a
```

Sample Code 2 (To show working of Symbol Table)

```
1#include <stdio.h>
2
3 struct node
4 {
5        int a ;
6        char b ;
7 } ;
8
9 void fun ( int x , char y )
10 {
11       int a = 5 , b = 6 ;
12
13       do {
14               a -= 1 ;
15       } while ( a > 0 ) ;
16
17       if ( a != 0 )
18       {
19               int y ;
20               a += 1 ;
21       }
22       int d ;
23       float z ;
24
25
26 }
27
28
```

Symbol Table

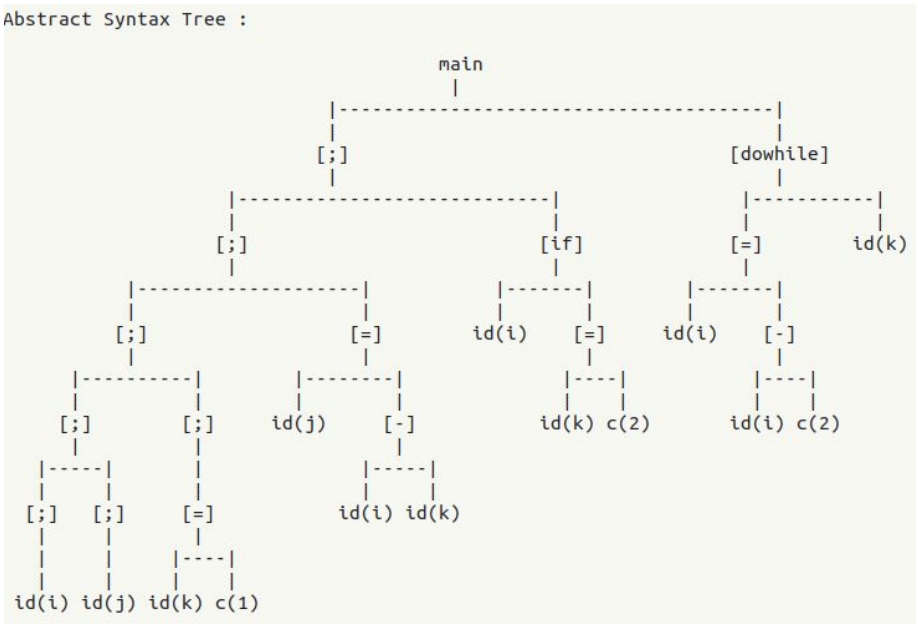
```
SYMBOL TABLE

Token   Type    Scope   Line No.
y       int     2       19
a               2       20
d       int     1       22
z       float   1       23
a               2       14
a       int     1       11
b               1       11
a       int     1       5
b       char    1       6
fun     void    0       9
x       int     0       9
y       char    0       9
```

Sample Code 3 (To show working of AST)

```c
int main()
{
  int i;
  int j;
  int k=1;
  j = i-k;
  if (i)
    k = 2;
  do
  {
    i = i - 2;
  } while (k);
}
```

Abstract Syntax Tree



Sample Code 4(To show working of ICG and code optimization)

```c
#include <stdio.h>

int main(int c , int d)
{
  a = 3 + 5*4 + 2;
  a = b * 3;
  c = d + 4;
  int var =3 , i = d;
  int a = c + d + 2;
  int b = f(a,b,c);
  // b = f(a,b,c);
  // f(a,b,c);

  if (x > y)
  {
    u = 1;
  }
  fact = u;

  do
  {
    u = 2;
  } while (z > w);

  if (x > y)
  {
    u = 1;
  }
}
```

Output of lex and yacc

```c
1#include <stdio.h>
2
3 int main ( int c , int d )
4 {
5 a = 3 + 5 * 4 + 2 ;
6 a = b * 3 ;
7 c = d + 4 ;
8 int var = 3 , i = d ;
9 int a = c + d + 2 ;
10 int b = f ( a , b , c ) ;
11
12
13
14 if ( x > y )
15 {
16   u = 1 ;
17 }
18 fact = u ;
19
20 do
21 {
22 u = 2 ;
23 } while ( z > w ) ;
24
25 if ( x > y )
26 {
27   u = 1 ;
28 }
29 }
30
```

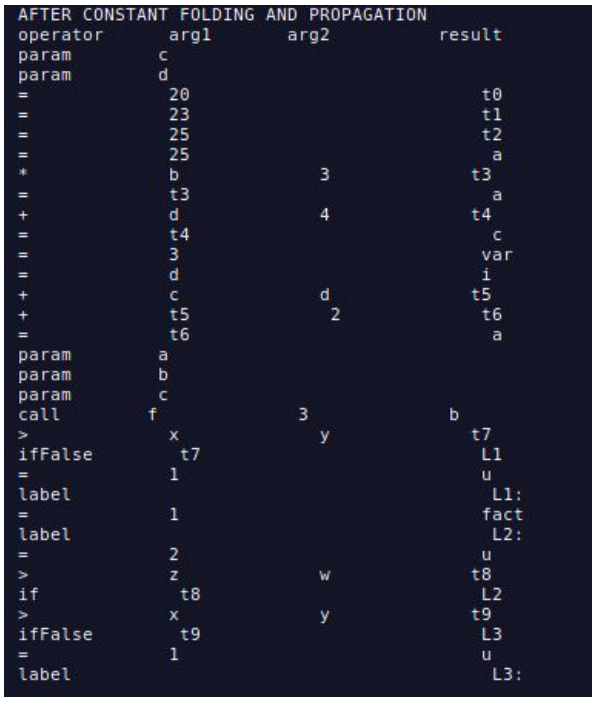
ICG

```
func   begin   main
param   c
param   d
t0  =   5   *   4
t1  =   3   +   t0
t2  =   t1  +   2
a   =   t2
t3  =   b   *   3
a   =   t3
t4  =   d   +   4
c   =   t4
var =   3
i   =   d
t5  =   c   +   d
t6  =   t5  +   2
a   =   t6
param   a
param   b
param   c
b   =   call f 3
t7  =   x   >   y
ifFalse t7 goto L1
u   =   1
L1:
fact    =   u
L2:
u   =   2
t8  =   z   >   w
if t8 goto L2
t9  =   x   >   y
ifFalse t9 goto L3
u   =   1
```

Quadruple Table Generation

```
QUADRUPLE TABLE
operator     arg1      arg2         result
param       c
param       d
*            5           4           t0
+            3           t0           t1
+            t1           2           t2
=            t2                       a
*            b           3           t3
=            t3                       a
+            d           4           t4
=            t4                       c
=            3                       var
=            d                       i
+            c           d           t5
+            t5           2          t6
=            t6                       a
param       a
param       b
param       c
call       f           3           b
>            x           y          t7
ifFalse       t7                     L1
=            1                       u
label                                L1:
=            u                      fact
label                                L2:
=            2                       u
>            z           w          t8
if            t8                     L2
>            x           y          t9
ifFalse       t9                     L3
=            1                       u
label                                L3:
```

Code Optimization



```
AFTER CONSTANT FOLDING AND PROPAGATION
operator      arg1       arg2          result
param         c
param         d
=              20                        t0
=              23                        t1
=              25                        t2
=              25                         a
*              b              3          t3
=              t3                        a
+              d              4          t4
=              t4                        c
=              3                        var
=              d                        i
+              c              d          t5
+              t5              2         t6
=              t6                        a
param         a
param         b
param         c
call         f              3          b
>              x              y         t7
ifFalse        t7                       L1
=              1                        u
label                                    L1:
=              1                        fact
label                                    L2:
=              2                        u
>              z              w         t8
if             t8                       L2
>              x              y         t9
ifFalse        t9                       L3
=              1                        u
label                                    L3:
```

9. CONCLUSIONS

Using lex and yacc, a compiler for C was created. Constructs like basic building blocks of the language (functions, declaration statements, assignment statements, etc) were handled, along with the main defined construct (do-while). The compiler went through the various phases of Compiler Design, ie, Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Optimisation. As a part of each stage, an intermediate part of the compiler was built (Symbol Table, Abstract Syntax Tree and Intermediate Code). Each of these components are required to compile code successfully.

10. FURTHER ENHANCEMENTS

- Functionality for for-loop and other data types can be implemented.
- The compiler can be constructed to recover from more kinds of errors
- Optimizations like dead code elimination and common sub expressions can be implemented using DAGs