Report on

"JavaScript Compiler"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Kara Akhila	**PES1201800311**
Nishtha Varshney	**PES1201801229**
Maknoor Shalini	**PES1201800253**

Under the guidance of

Preet Kanwal
Assistant 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 - 1
INTRODUCTION

The Mini Compiler has been made for the language **JavaScript.**The constructs that have been focused on are ‘do while’ and ‘for’ loop.The optimizations that were implemented are as follows:

- Elimination of dead code/unreachable code
- Implementation Copy Propagation
- Implementation of Constant folding
- Implementation of Constant Propagation

Syntax and semantic errors were taken care of and Syntax error recovery has been implemented using Panic Mode Recovery in lexer.

The Screenshot of the sample Input and target code output as follows:

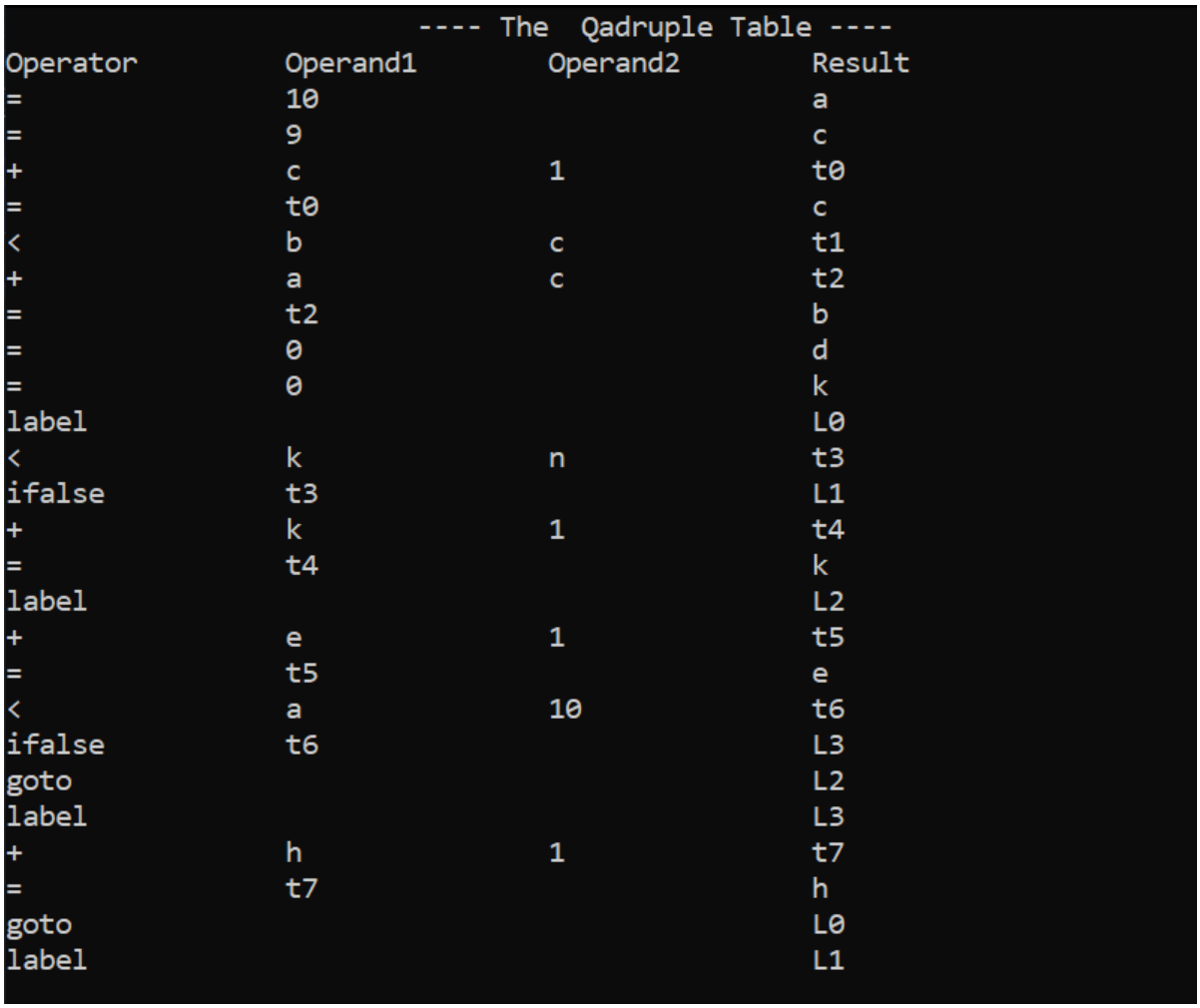
Input 1:

```
var a;
a=10;
var c=9;
c++;
b<c;
var b=a+c;
var d=0;
for(k=0;k<n;k++)
{
    do{
        e=e+1;
    }while(a<10);
    h++;
}
```

Target output: Symbol table

```
                    ----- Symbol Table -----
Line number     Var name        Value           Type            Scope
5               a               10              integer         0
5               c               10              integer         0
5               b               20              integer         0
6               d               0               integer         0
7               k               1               integer         0
7               n               0               integer         0
10              e               1               integer         2
11              a               0               integer         1
12              h               1               integer         1
```

Quadruple Table



---- The Qadruple Table ----

Operator	Operand1	Operand2	Result
=	10		a
=	9		c
+	c	1	t0
=	t0		c
<	b	c	t1
+	a	c	t2
=	t2		b
=	0		d
=	0		k
label			L0
<	k	n	t3
ifalse	t3		L1
+	k	1	t4
=	t4		k
label			L2
+	e	1	t5
=	t5		e
<	a	10	t6
ifalse	t6		L3
goto			L2
label			L3
+	h	1	t7
=	t7		h
goto			L0
label			L1

CHAPTER - 2
ARCHITECTURE OF LANGUAGE

In this Mini Compiler , the following aspects of the JavaScript language syntax have been covered:

- Constructs like ‘do-while’ and ‘for’
- Nested loops
- Single and multi-line comments.
- Integer and float data types.

Depending on the type of error, different error messages are shown. The yyerror() function is used to handle syntax errors, while a call to a function that searches the symbol table for a specific identifier is used to handle semantic errors. As part of the error message, the line number is shown.

Input:

```
var a;
a=2
```

Output:

```
shalini@DESKTOP-HB38583:~/CD/Semantic$ flex lex.l && bison -d -y yacc.y && gcc y.tab.c lex.yy.c -ll && ./a.out <inp1.txt

Error in line 1, syntax error, unexpected $end, expecting T_SEMICOL
```

```
shalini@DESKTOP-HB38583:~/CD/ICG$ flex lex.l && bison -d -y yacc.y && gcc y.tab.c lex.yy.c -ll && ./a.out <ip1
Floating point exception
```

CHAPTER - 3
LITERATURE SURVEY

1. **Introduction to YACC:**
   https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf

2. **Intermediate Code Generation**
   https://www.gnu.org/software/bison/manual/bison.html

3. **Code Optimization**
   https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/code.opt.pdf
   https://ccsuniversity.ac.in/bridge-library/pdf/MCA-0705-II-UNIT-5-Code-Optimization-&-Code-Generation.pdf

CHAPTER - 4
CONTEXT FREE GRAMMAR

REGEX:

```
digit   [0-9]
letter  [a-zA-Z]
id      ({letter}|\$|\_)({letter}|{digit}|\$|\_)*
digits  {digit}+
opFraction  (\.{digits})?
opExponent  ([Ee][+-]?{digits})?
number  {digits}{opFraction}{opExponent}
string  \"({letter}|{digit})+\"
start   \/\*
end \*\/
```

GRAMMAR

start : prog{printf("Valid Input\n");printf("\t\t\t----- Symbol Table -----\n");print_list(head);};

prog :prog S
|
;

S :D
| arr T_SEMICOL
| A T_SEMICOL
| T_FOR T_OP A T_SEMICOL cond T_SEMICOL una T_CP T_OFB prog T_CFB

```
        | T_DO T_OFB  S  T_CFB T_WHILE T_OP cond T_CP
          T_SEMICOL
        | E T_SEMICOL
        | cond T_SEMICOL
        | una T_SEMICOL
        | T_DOC T_OP E T_CP T_SEMICOL
        | T_CON T_OP E T_CP T_SEMICOL
        ;
D       : T L T_SEMICOL
        | T T_ID T_EQUAL T_NEW T_ARRAY T_OP T_NUM T_CP
          T_SEMICOL
        ;

arr     : T_ID T_OSB T_NUM T_CSB T_EQUAL T_NUM
        | T_ID T_OSB T_NUM T_CSB T_EQUAL T_STR
        ;
T       : T_VAR
        | T_CONST
        ;
L       : X T_C L
        | X
        ;
X       : T_ID{search_and_replace_or_add(line,var_name," "," ",scope);}
        | A
        ;
A       : T_ID T_EQUAL E
          {search_and_replace_or_add(line,var_name,value,type,scope);}
        ;
cond    : E rel E
        ;
una     : T_ID T_I
        | T_ID T_D
        | T_I T_ID
        | T_D T_ID
        ;
rel     : T_LT
        | T_GT
```

```
        | T_GTE
        | T_LTE
        | T_EQ
        | T_NEQ
        ;
E       : E T_PL T
        | E T_MI T
        | T
        ;
T       : T T_MUL G
        | T T_DIV G
        | G
        ;
G       :
T_ID{search_and_replace_or_add(line,var_name,value,type,scope);}
        | T_NUM
        | T_STR
        ;
```

FOR Optimisation

REGEX:

```
[0-9]+                          {yylval.rvalues.value = strdup(yytext); return T_NUM;}
label                           {yylval.rvalues.value = strdup(yytext); return T_LABEL;}
iffalse                         {yylval.rvalues.value = strdup(yytext); return T_FALSE;}
iftrue                          {yylval.rvalues.value = strdup(yytext); return T_TRUE;}
goto                            {yylval.rvalues.value = strdup(yytext); return T_GOTO;}
l[0-9]+                         {yylval.rvalues.value = strdup(yytext); return T_LVAR;}
t[0-9]+                         {yylval.rvalues.value = strdup(yytext); return T_TVAR;}
([a-zA-Z]|\_)([a-zA-Z0-9]|\_)*  {yylval.rvalues.value = strdup(yytext); return T_VAR;}
(\[|\])                         {return(yytext[0]);}
\"([^"]|\\\")*\"|\'([^']|\\\')*\'   {yylval.rvalues.value = strdup(yytext); return T_STR;}
=                               {yylval.rvalues.value = strdup(yytext); return T_ASSIGN;}
(&&|\|\|)                       {yylval.rvalues.value = strdup(yytext); return T_BOOLOP;}
(\<|\>|\=)\=?                   {yylval.rvalues.value = strdup(yytext); return T_LOGOP;}
(\+|\-|\*|\/)                   {yylval.rvalues.value = strdup(yytext); return T_MATHOP;}
[\n]                            {yylval.rvalues.value = strdup(yytext); return T_DELIM;}
[\t ]                           {}
```

GRAMMAR

start : prog

prog : block Y;

Y : prog
|
;

block :S X
;

X : T_DELIM
|
;

S : T_TVAR T_ASSIGN value_prod
| T_VAR T_ASSIGN T_VAR {add_node($1.value,$3.value,2);}
| T_VAR T_ASSIGN value_constants{check_value(&$1.value, &$2.value, &$3.value);printf("%s %s %s\n", $1.value, $2.value, $3.value);}
| T_VAR T_ASSIGN value_constants operator T_VAR {check_value(&$3.value, &$4.value, &$5.value);printf("%s %s %s %s %s\n",$1.value, $2.value, $3.value, $4.value, $5.value);}
| T_TVAR T_ASSIGN T_NUM operator no_num {check_value(&$1.value, &$3.value, &$5.value); printf("%s %s %s %s %s\n", $1.value, $2.value, $3.value, $4.value, $5.value);}
| T_TVAR T_ASSIGN no_num operator T_NUM {check_value(&$1.value, &$3.value, &$5.value); printf("%s %s %s %s %s\n", $1.value, $2.value, $3.value, $4.value, $5.value);}
| T_TVAR T_ASSIGN T_NUM operator T_NUM {printf("%s = %s\n", $1.value, constant_fold($3.value, $4.value, $5.value));}
| T_TVAR T_ASSIGN no_num operator no_num {check_value(&$1.value, &$3.value, &$5.value); printf("%s %s %s

```
      %s %s\n", $1.value, $2.value,$3.value, $4.value, $5.value);}
    | T_FALSE value_constants T_GOTO T_LVAR
     {check_value(&$1.value, &$2.value, &$3.value);printf("%s %s %s
      %s\n", $1.value, $2.value, $3.value, $4.value);}
    | T_TRUE value_constants T_GOTO T_LVAR
     {check_value(&$1.value, &$2.value, &$3.value);printf("%s %s %s
       %s\n", $1.value, $2.value, $3.value, $4.value);}
    | T_GOTO T_LVAR  {printf("%s %s\n", $1.value, $2.value);}
    | T_LABEL T_LVAR {printf("%s %s\n", $1.value, $2.value);}
    | T_TVALVEC T_ASSIGN T_TVAR {check_value(&$1.value,
     &$2.value, &$3.value);printf("%s %s %s\n", $1.value, $2.value,
     $3.value);}
    | T_TVAR '[' tvar_tnum ']' T_ASSIGN tvar_tnum
     {check_value(&$1.value, &$3.value, &$6.value);printf("%s[%s] %s
      %s\n", $1.value, $3.value, $5.value, $6.value);}
    | T_TVAR T_ASSIGN value_constants '[' value_constants ']'
      {check_value(&$1.value, &$3.value, &$5.value);printf("%s %s
       %s[%s]\n", $1.value, $2.value, $3.value, $5.value);}


no_num    : T_VAR {$$.value = $1.value;}
          | T_TVAR {$$.value = $1.value;}

tvar_tnum  : T_NUM {$$.value = $1.value;}
           | T_TVAR {$$.value = $1.value;}

value_constants    : T_NUM {$$.value = $1.value;}
                   | T_STR {$$.value = $1.value;}
                   | T_TVAR {$$.value = $1.value;}

value_prod   : T_NUM {add_node($<rvalues>-1.value, $1.value, 0);}
             | T_STR {add_node($<rvalues>-1.value, $1.value, 1);}
             | T_VAR {add_node($<rvalues>-1.value, $1.value, 2);}

operator    : T_BOOLOP {$$.value = $1.value;}
            | T_LOGOP {$$.value = $1.value;}
            | T_MATHOP {$$.value = $1.value;}
```

CHAPTER - 5
DESIGN STRATEGY

1. LEX FILE - Generating Tokens

2. YACC FILE - Implementation of Grammar file

3. SYMBOL TABLE GENERATION
   The symbol table is built using a linked list. The variable name, value, scope, line number, and data type are all displayed in the final production. To produce the symbol table, we generated three functions.
   - Make_node() - This function helps us to create a node whenever it encounters a variable.
   - Search_replace_add() - This function will first search the symbol table for the variable if it exists . It updates the values of the variable,else it will create a node using Make_node() function.
   - Print_list() - This function prints the symbol table.

4. INTERMEDIATE CODE GENERATION
   Intermediate code that is generated in this phase has been represented in quadruple format. Array of type struct has been used to store the generated code.

5. CODE OPTIMIZATION

A data structure called SYMTAB holds the details of each assignment.

6. ERROR HANDLING
   Syntax error: If the token returned does not satisfy the grammar, then yyerror() is used to display the syntax error along with the line number.

CHAPTER - 6
IMPLEMENTATION DETAILS

1. LEX FILE

```
var {return VAR;}
let return LET;
const   return CONST;
for return FOR;
do  return DO;
while   return WHILE;
document.write return DOC;
console.log return CON;
{id}    {strcpy(var_name,yytext);return ID;}
{number}    {yylval = atoi(yytext);strcpy(type,"integer");return NUM;}
{string}    {strcpy(type,"string");return STR;}
":" return T_COL;
";" return T_SEMICOL;
">" return T_GT;
"<" return T_LT;
">="    return T_GTE;
"<="    return T_LTE;
"=="    return T_EQ;
"!="    return T_NEQ;
```

2. YACC FILE

   This is a snapshot of the small part of the code in the yacc file.

```
S   :D
    |arr T_SEMICOL
    |A T_SEMICOL
    |T_FOR T_OP A T_SEMICOL cond T_SEMICOL una T_CP  T_OFB prog T_CFB
    |T_DO T_OFB  S  T_CFB T_WHILE T_OP cond T_CP T_SEMICOL
    |E T_SEMICOL
    |cond T_SEMICOL
    |una T_SEMICOL
    |T_DOC T_OP E T_CP T_SEMICOL
    |T_CON T_OP E T_CP T_SEMICOL
    ;
```

3. SYMBOL TABLE

The following snapshot shows the structure declaration for symbol table:

```
typedef struct token_node
{
    int line;
    char* var_name;
    char* value;
    char* type;
    int scope;
    struct token_node* next;
} token_node;
```

These are the functions used to generate the symbol table:

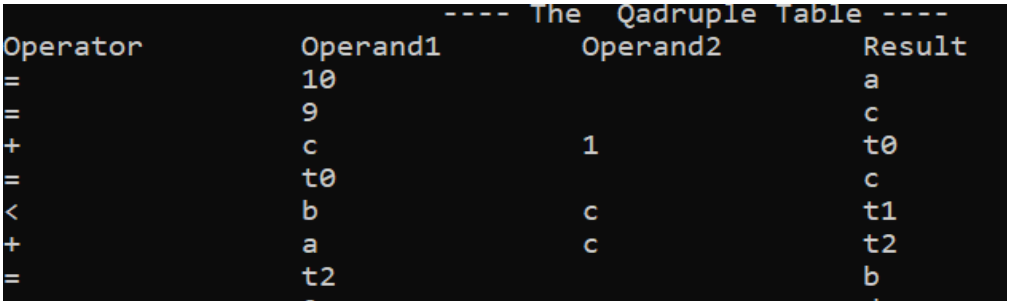
```
token_node* make_node(int line, char* var_name, char* value, char* type,int scope);
void search_and_replace_or_add(int line,char* var_name,char* value,char* type,int scope);
void print_list();
```

4. INTERMEDIATE CODE GENERATION

Quadruple format has been used to represent Intermediate code.
A struct has been used to define each entry.

```
struct  quadruple
  {
    char operator[10];
    char operand1[10];
    char operand2[10];
    char result[10];
  } quad[50];
```

The quadruple table looks as below.



```
                    ---- The  Qadruple Table ----
Operator            Operand1            Operand2            Result
=                   10                                      a
=                   9                                       c
+                   c                   1                   t0
=                   t0                                      c
<                   b                   c                   t1
+                   a                   c                   t2
=                   t2                                      b
```

5. CODE OPTIMIZATION
   (Elimination of Dead code, Constant folding, Constant propagation and Copy Propagation)

   The following snapshot shows the structure declaration for SYMTAB. This structure holds the details of all the variables.

```
struct SYMTAB {
        struct SYMTAB *next;
        struct SYMTAB *prev;
        char *name;
        char *val;
        int type;
};
```

These are the functions used to optimise the code :

```
void check_value(char **, char **, char **);
struct SYMTAB * find_node(char *);
void add_node(char *, char *, int);
char * constant_fold();
```

6. ERROR HANDLING

The following snapshot shows the error handling function for syntax errors:

```
void yyerror(char *s){printf("\nError in line %d, %s\n",line,s);return;}
```

BUILD AND RUN THE PROGRAM

The following screenshot displays what commands need to be executed to build and run the program:

```
$ flex lex.l
$ bison -d -y par.y
$ gcc y.tab.c lex.yy.c -w -ll -fdce
$ ./a.out <inp1
```

The above commands need to be executed on the terminal which is inside the project folder that contains the code for the entire compiler.

CHAPTER - 7
RESULTS AND SHORTCOMINGS

The small scale compiler worked in this task turns out consummately for the 'do-while' and ‘for’ keeping in mind that constructs of Javascript language. Our compiler can be executed in various stages by building and running the code isolated in the different folders.The final code output displays the the symbol table along with the quadruple table is printed without optimisations.Finally, the symbol table and the intermediate code after optimisations is displayed after the quadruples table. This is for inputs with no errors. But in case of erroneous inputs, the token generation is stopped on error encounter and the corresponding error message is displayed.

This mini-compiler has the following shortcomings:
• User defined functions are not handled.
• Importing libraries and calling library functions is not taken care of.
• Constructs other than ‘do-while’ and ‘for’ have not been added in the compiler program.

CHAPTER - 8
SNAPSHOTS

Test Case 1 (Correct Input)

Input 1 :

```
var a;
a=10;
var c=9;
c++;
b<c;
var b=a+c;
var d=0;
for(k=0;k<n;k++)
{
    do{
        e=e+1;
    }while(a<10);
    h++;
}
```

Output : (Symbol Table and Quadruple Table)

```
                    ----- Symbol Table -----
Line number     Var name        Value           Type            Scope
5               a               10              integer         0
5               c               10              integer         0
5               b               20              integer         0
6               d               0               integer         0
7               k               1               integer         0
7               n               0               integer         0
10              e               1               integer         2
11              a               0               integer         1
12              h               1               integer         1
```

```
                        ---- The  Qadruple Table ----
Operator        Operand1        Operand2        Result
=               10                              a
=               9                               c
+               c               1               t0
=               t0                              c
<               b               c               t1
+               a               c               t2
=               t2                              b
=               0                               d
=               0                               k
label                                           L0
<               k               n               t3
ifalse          t3                              L1
+               k               1               t4
=               t4                              k
label                                           L2
+               e               1               t5
=               t5                              e
<               a               10              t6
ifalse          t6                              L3
goto                                            L2
label                                           L3
+               h               1               t7
=               t7                              h
goto                                            L0
label                                           L1
```

Input for Optimization of code :

```
x = y
z = 3 + y
t0 =  10
n  = t0
t1 =  0
c  = t1
t2 =  0
i  = t2
label l0
t3 =  i
t4 =  n
t5 = t3 < t4
iffalse t5 goto l1
t9 =  1
c  = t9
t6 =  i
t7 =  1
t8 = t6 + t7
goto l0
label l1
a = 10
b = 20
t10 = 4*j
t11[t10] = 100
label l2
```

Output :

```
z = 3 + y
n = 10
c = 0
i = 0
label l0
t5 = i < n
iffalse t5 goto l1
c = 1
t8 = i + 1
goto l0
label l1
a = 10
b = 20
t10 = 4 * j
t11[t10] = 100
label l2
```

Test Case 2 (Syntax Error)

```
var a;
a=10;
var c=9;
c++;
b<c
var b=a+c;
var d=0;
```

```
nishtha@Nishtha:~/cd/ICG$ flex lex.l && bison -d -y yacc.y && gcc y.tab.c lex.yy.c -w -ll -fdce && ./a.out <ip

Error in line 5, syntax error
nishtha@Nishtha:~/cd/ICG$
```

CHAPTER - 9
CONCLUSIONS

- This is a python mini-compiler that uses lex and yacc files to take in a Javascript program and test it according to the context free grammar written.
- Regular Expressions are written to generate the tokens.
- Symbol table is created to store the information about the identifiers.
- Quadruple table is generated and the data structure used for optimisation is Quadruples. The optimisation techniques used are constant propagation, copy propogation, dead code elimination and constant folding.
- Error handling and recovery implemented take care of erroneous inputs.

CHAPTER - 10
FURTHER ENHANCEMENTS

With a few tweaks, this mini-compiler can be expanded into a full Javascript compiler. User defined functions can be handled, as well as the features of importing libraries and calling library functions. Other data types, such as sets, tuples, dictionaries, and structures other than ‘do-while' and ‘for' such as ‘if-else' can be included in the compiler program. The performance can be improved and made to look more appealing. Using different data types, functions, or methods will increase the program's overall performance and tempo.