Report on

“Mini-Compiler for Javascript ”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Raksha Ramesh	**PES1201800345**
Swanuja Maslekar	**PES1201800369**
Vidish Raj	**PES1201800223**

Under the guidance of

Madhura V
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

Introduction

The mini-compiler was built for Javascript. The project was split into two phases - phase1 and phase2. In phase1, the implementation was focussed on the lexical phase, syntax phase and semantic phase. In this phase of the project, regular grammar was used to define rules and generate tokens. Comment removal, data conversion and other small details were handled. The symbol table was generated to keep track of identifiers in the code, along with the line used, scope, etc.
(refer to snapshots below).
The second phase focussed on intermediate code generation and optimization. It involved evaluation of expression and updation of the symbol table, dead code elimination, common subexpression elimination and so on. Various different test cases were assessed on the compiler, screenshots of which can be found below.

Architecture of language

The syntax and semantics that were handled in our javascript compiler were:

- Conversion to int and real valued numerical data and other necessary conversions.
- All operators and punctuation characters are handled by the compiler.
- All identifiers are tokenized and printed on the symbol table along with other details like the line used, scope, value,etc.
- Comments were stripped off, non-matching tokens are reported and identifier length was restricted.
- Verification of tokens to form a sentence and construction of the abstract tree.
- Error handling and recovery was done.
- Other semantics like type checking, variable declaration , loops and errors were handled.
- Three address code was generated and dead code/unreachable code is eliminated.
- Optimization was performed on the three address code that was generated.

Context free grammar

The context free grammar used to implement the tokenization of the JavaScript input was as follows:

```
start: seqOfStmts;
seqOfStmts: statement seqOfStmts | statement;
anyOperator: T_LCG | T_LOP | T_OP1 | T_OP2 | T_OP3;
terminator: ';' | '\n';
statement: declare terminator|expr terminator|for |if | while|
'{'{scope[stop++]=sid++;} seqOfStmts '}' {stop--;} | T_CONSOLE
'(' T_STR ')' | T_DOCUMENT '(' T_STR ')' ;
id:  T_ID  {mkentr(0,identifier,scope[stop-1]);printf("updating
:%s scope:%d\n",identifier,scope[stop-1]);};
idV:     T_ID     {chkentr(identifier);printf("checking    :%s
scope:%d\n",identifier,scope[stop-1]);};
```

```
assign: '=' | T_SHA;
expr: id assign expr | value | ;
value: unit anyOperator value |unit;
unit:   idV   |   T_OP4   idV   |   idV   T_OP4   |   T_STR
{add_type_name(identifier,           1);}|           T_NUM
{add_type_name(identifier, 0);}| '(' list ')'| func | '[' list
']';
func: idV '(' list ')';
list: expr ',' list | expr;
declare: T_VAR mulDecl | T_LET mulDecl;
mulDecl:  id  |id  ','  mulDecl|id  '='  expr|id  '='  expr  ','
mulDecl;
varOperator: T_VAR | T_LET | ;
for:  T_FOR  '('  varOperator  list  ';'  list  ';'  list  ')'
statement;
if: T_IF '(' expr ')' statement ifelse;
while: T_WHILE '(' expr ')' statement;
ifelse: T_ELSE statement|;
```

Design details

Symbol table:
The symbol table was designed keeping in mind the syntax of Javascript. The generated tokens had to be represented in the symbol table along with its important attributes such as name, scope, type, value, declared line and the last line used. The symbol table was designed to print along with the strings and the numbers in the input. In phase1, all the functions related to the creation of the symbol table, making an entry, checking an entry, printing the symbol table and so on were entered in a file named sbtls.h.

Intermediate Code Generation:
The parser takes an input from the input file and generates an icg.txt which contains the intermediate code generated in quadruple format. It employs various data structures to define the structure of each variable and facilitates the storing of the abstract syntax tree and the intermediate code generated for all the variables in the LHS of the grammar rules. This is done with the help of production rules.

Error handling:
The scanner was designed to handle various different errors. Unterminated comments would be reported. Lexical errors like lengthy identifiers, improper strings and invalid characters would be reported as well.

Implementation details

Symbol table-:
Context free grammar was used to define syntax rules of javascript. The tokens were generated based on these rules. The symbol table implementation was done while keeping the design in mind. The symbol table was implemented using linked lists. A structure was composed with integer values for the type, scope, line used

and line it was declared on. A pointer to the structure was kept to implement a linked list to hold all the token values and data. Three separate linked lists were created to hold identifiers, strings and numerical values. The linked list was traversed to print the table in the order variables were encountered. Separate counters were used to keep track of the scope of the variables.

Intermediate Code Generation:

For the implementation of the intermediate code generation, the same grammar used in the previous phase was modified to include production rules.
A couple of data structures were defined to make the storing of the outputs easy.

The following were the data structures used:

```
struct {char *code,*ast;int next;} stt;
struct {char *code,*ast;int idn;} eq;
struct {int dt[4];} dt;
struct {int idn,off;char *code,*ast;} ls;
```

The LHS of the grammar rules were assigned to their respective data structure:

```
%type<stt> seq statement for if while
%type<eq> expr unit defn anyopl anyoph rhsl rhsh
%type<ls> list
%type<dt> lhs lhsv edt
```

Production rules were added for each grammar rule so that `seq.code` contains the required ICG in Quadruple format.
Each production rule makes sure that the LHS of the concerning grammar rule `LHS.code` contains the ICG in quadruple format for that section of the code.
For example, for the grammar rule: `while: T_WHILE '(' expr ')' statement;`
`while.code` will have the quadruple code for the entire while loop.

Here is the actual code:

```
while: T_WHILE edt '(' expr ')'
{$2.dt[0]=lbl++;$2.dt[1]=lbl++;}
          statement {char *a,*b,*c;
          sprintf(bbuf,"label\t\t\t \t\t\t
\t\t\tl%d\n",$2.dt[0]);
          a = ap(strdup(bbuf),$4.code);
          sprintf(bbuf,"iffalse\t\t\tt%d\t\t\t
\t\t\tl%d\n",$4.idn,$2.dt[1]);
          b=ap(strdup(bbuf),$7.code);
          sprintf(bbuf,"goto\t\t\t \t\t\t
\t\t\tl%d\nlabel\t\t\t \t\t\t \t\t\tl%d\n",$2.dt[0],$2.dt[1]);
          $$.code=ap3(a,b,strdup(bbuf));
          a=ap3(strdup("while("),$4.ast,strdup(")"));
          $$.ast=ap(a,$7.ast);
          };
```

The ast is also appended similarly.

[Note: ap() and ap3() are user defined string manipulation functions that concatenate 2 and 3 string arguments respectively]

Error handling:
If there is a syntax error, the parser will stop at that line. However, small errors like missing semicolon and extra \n are ignored by the parser and it continues parsing.

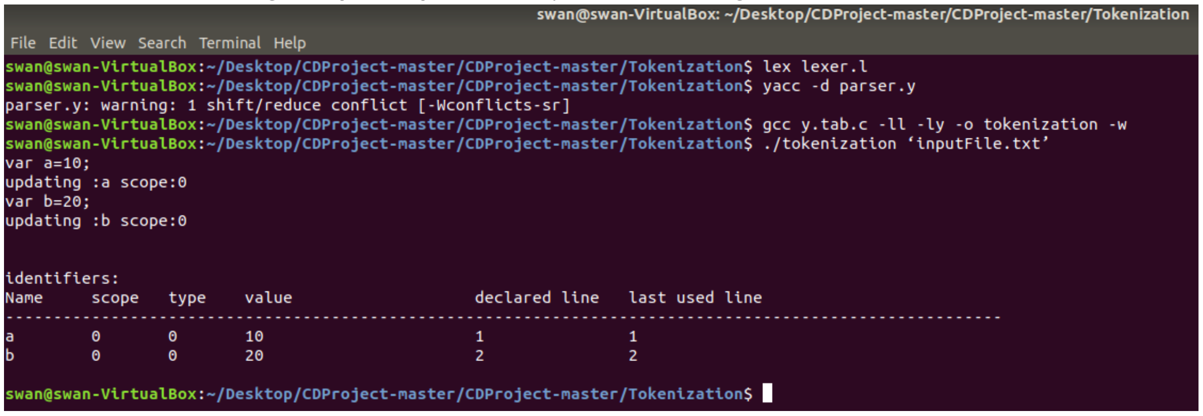
Result and possible shortcomings

The input files that were parsed by our compiler generated tokens and symbol table as planned. The intermediate code generation and error handling was as desired. The compiler passed the test cases that it was tested against.
The compiler isn't capable of handling other loops like switch and for.

Screenshots

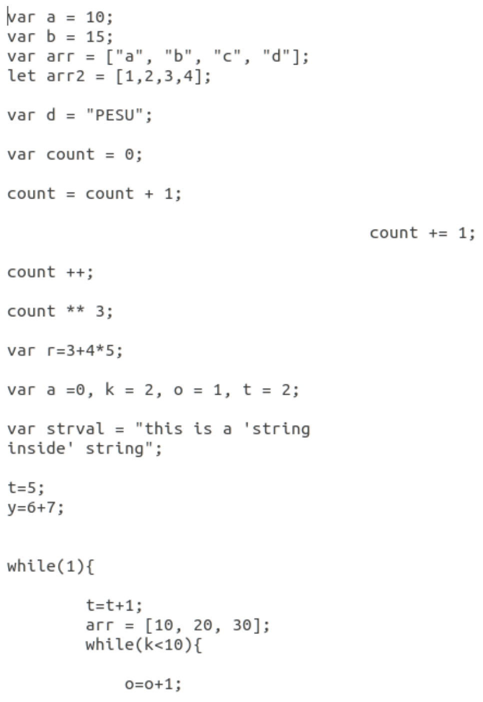
Phase 1

1. Screenshot showing simple input and symbol table generation


swan@swan-VirtualBox: ~/Desktop/CDProject-master/CDProject-master/Tokenization
File Edit View Search Terminal Help

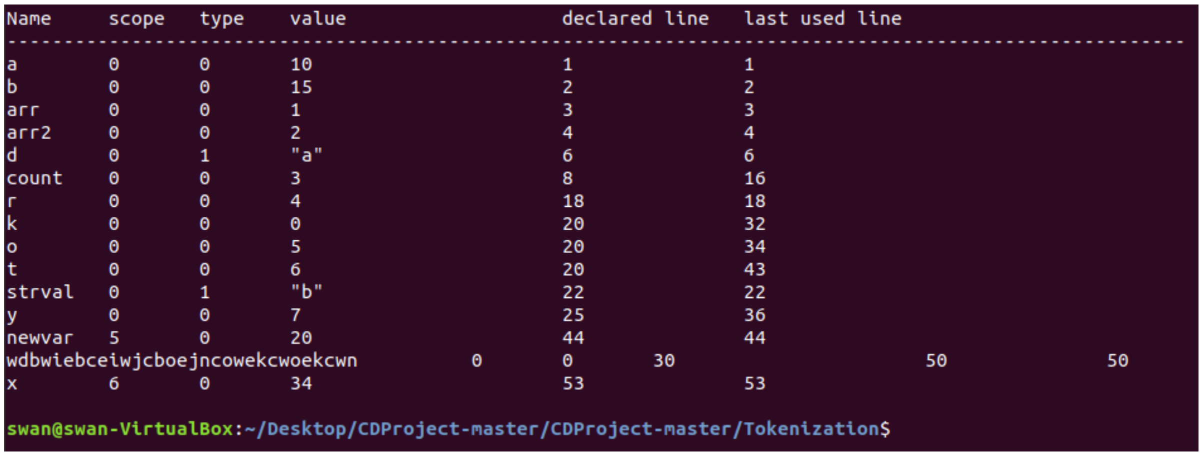
```
swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$ lex lexer.l
swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$ yacc -d parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$ gcc y.tab.c -ll -ly -o tokenization -w
swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$ ./tokenization 'inputFile.txt'
var a=10;
updating :a scope:0
var b=20;
updating :b scope:0


identifiers:
Name      scope   type    value                  declared line   last used line
-------------------------------------------------------------------------------------------------------------
a         0       0       10                     1               1
b         0       0       20                     2               2

swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$ 
```

2. Sample input sent



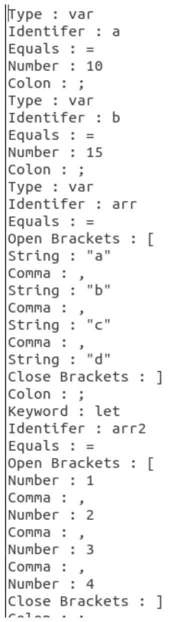
```
var a = 10;
var b = 15;
var arr = ["a", "b", "c", "d"];
let arr2 = [1,2,3,4];

var d = "PESU";

var count = 0;

count = count + 1;

                                     count += 1;

count ++;

count ** 3;

var r=3+4*5;

var a =0, k = 2, o = 1, t = 2;

var strval = "this is a 'string
inside' string";

t=5;
y=6+7;


while(1){

        t=t+1;
        arr = [10, 20, 30];
        while(k<10){

            o=o+1;
```

Symbol table generated



```
Name      scope   type    value                      declared line   last used line
------------------------------------------------------------------------------------------------------------------------
a          0       0       10                         1               1
b          0       0       15                         2               2
arr        0       0       1                          3               3
arr2       0       0       2                          4               4
d          0       1       "a"                        6               6
count      0       0       3                          8               16
r          0       0       4                          18              18
k          0       0       0                          20              32
o          0       0       5                          20              34
t          0       0       6                          20              43
strval     0       1       "b"                        22              22
y          0       0       7                          25              36
newvar     5       0       20                         44              44
wdbwiebceiwjcboejncowekcwoekcwn          0       0       30                         50              50
x          6       0       34                         53              53

swan@swan-VirtualBox:~/Desktop/CDProject-master/CDProject-master/Tokenization$
```

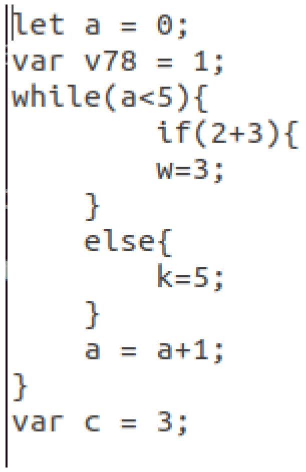
Tokens generated



```
Type : var
Identifer : a
Equals : =
Number : 10
Colon : ;
Type : var
Identifer : b
Equals : =
Number : 15
Colon : ;
Type : var
Identifer : arr
Equals : =
Open Brackets : [
String : "a"
Comma : ,
String : "b"
Comma : ,
String : "c"
Comma : ,
String : "d"
Close Brackets : ]
Colon : ;
Keyword : let
Identifer : arr2
Equals : =
Open Brackets : [
Number : 1
Comma : ,
Number : 2
Comma : ,
Number : 3
Comma : ,
Number : 4
Close Brackets : ]
```

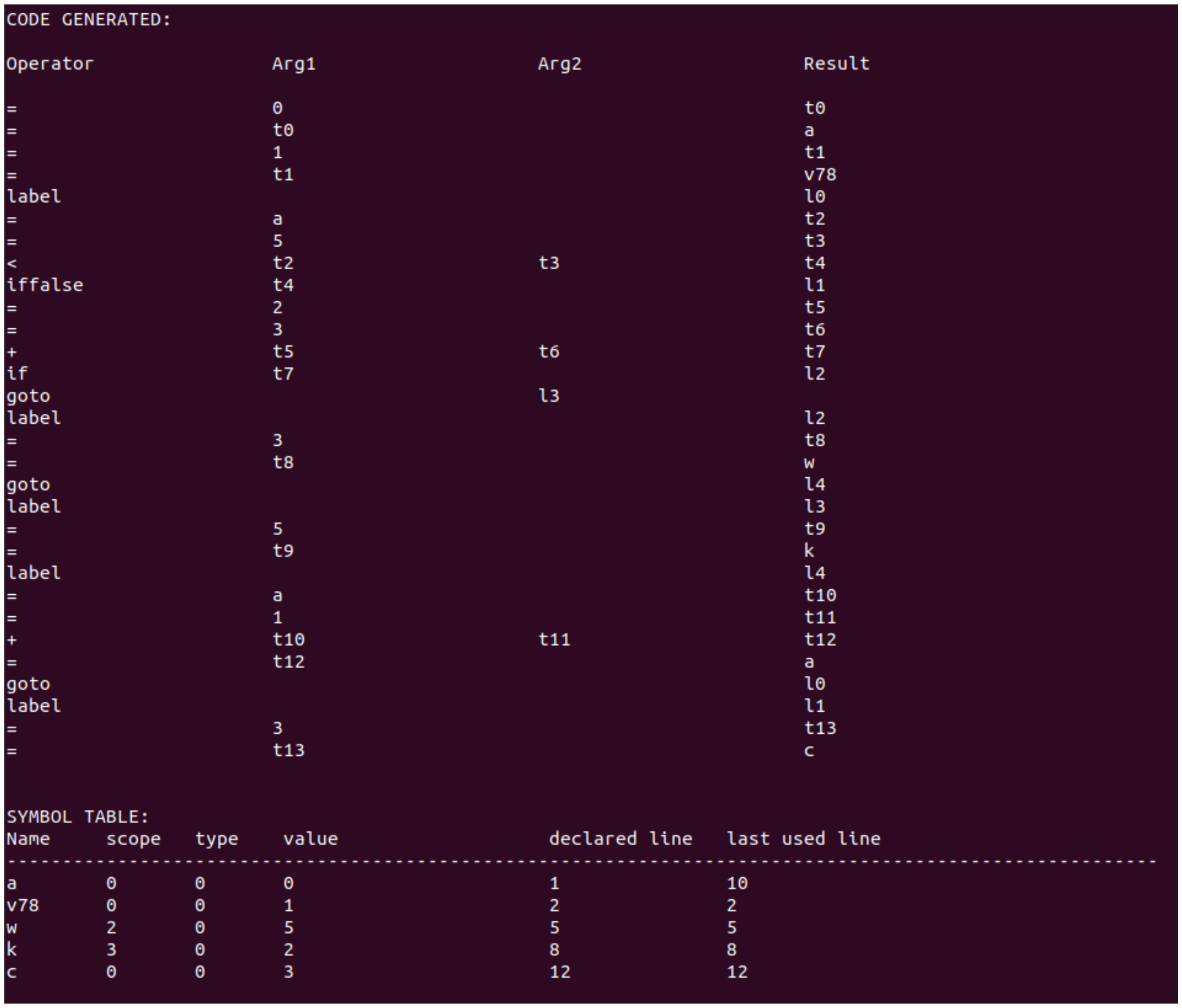
Phase2

Sample input



```
let a = 0;
var v78 = 1;
while(a<5){
        if(2+3){
        w=3;
    }
    else{
        k=5;
    }
    a = a+1;
}
var c = 3;
```

Code generated

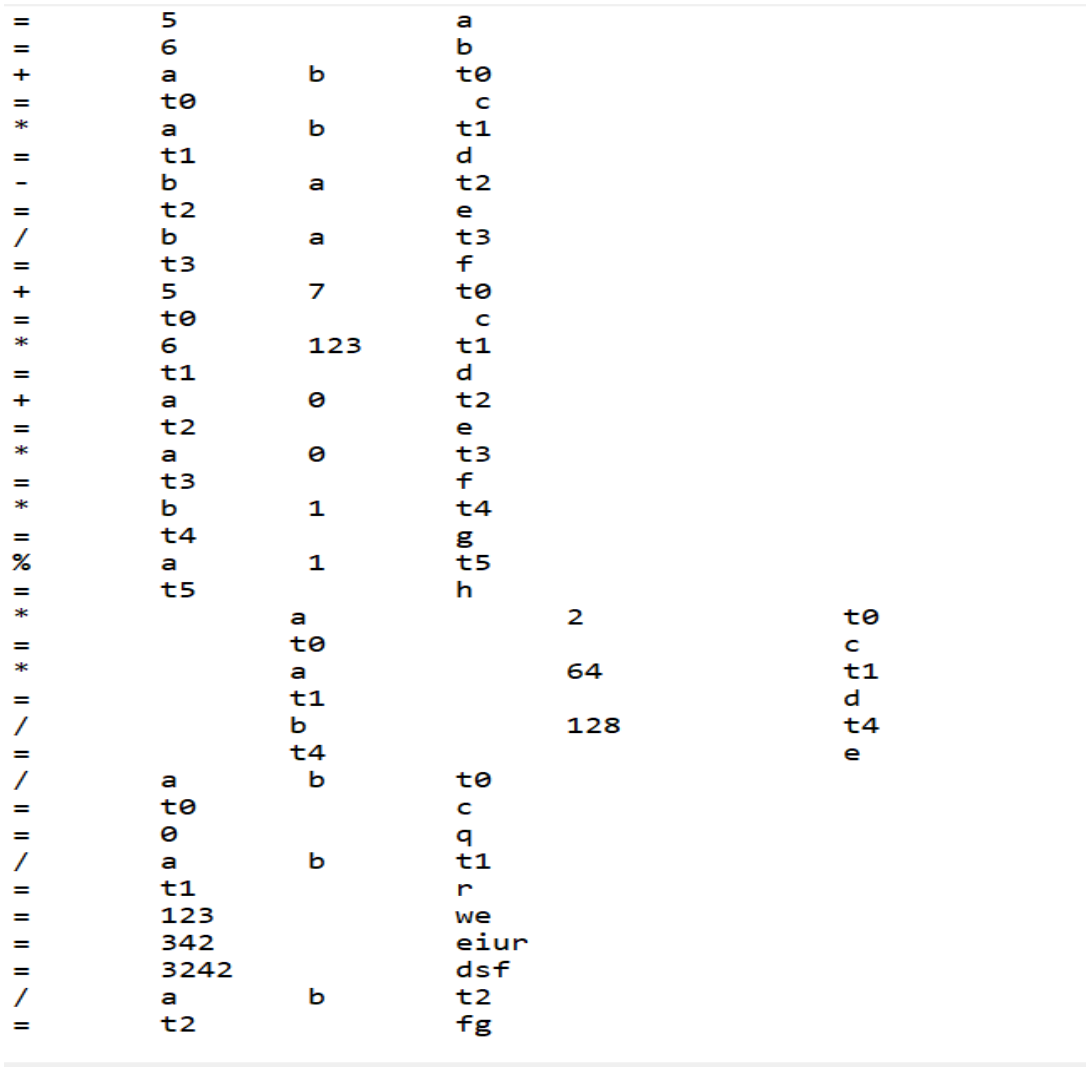


```
CODE GENERATED:

Operator               Arg1                   Arg2                   Result

=                      0                                             t0
=                      t0                                            a
=                      1                                             t1
=                      t1                                            v78
label                                                                l0
=                      a                                             t2
=                      5                                             t3
<                      t2                     t3                     t4
iffalse                t4                                            l1
=                      2                                             t5
=                      3                                             t6
+                      t5                     t6                     t7
if                     t7                                            l2
goto                                          l3
label                                                                l2
=                      3                                             t8
=                      t8                                            w
goto                                                                 l4
label                                                                l3
=                      5                                             t9
=                      t9                                            k
label                                                                l4
=                      a                                             t10
=                      1                                             t11
+                      t10                    t11                    t12
=                      t12                                           a
goto                                                                 l0
label                                                                l1
=                      3                                             t13
=                      t13                                           c


SYMBOL TABLE:
Name     scope   type    value                  declared line   last used line
------------------------------------------------------------------------------------------------------
a        0       0       0                      1               10
v78      0       0       1                      2               2
w        2       0       5                      5               5
k        3       0       2                      8               8
c        0       0       3                      12              12
```

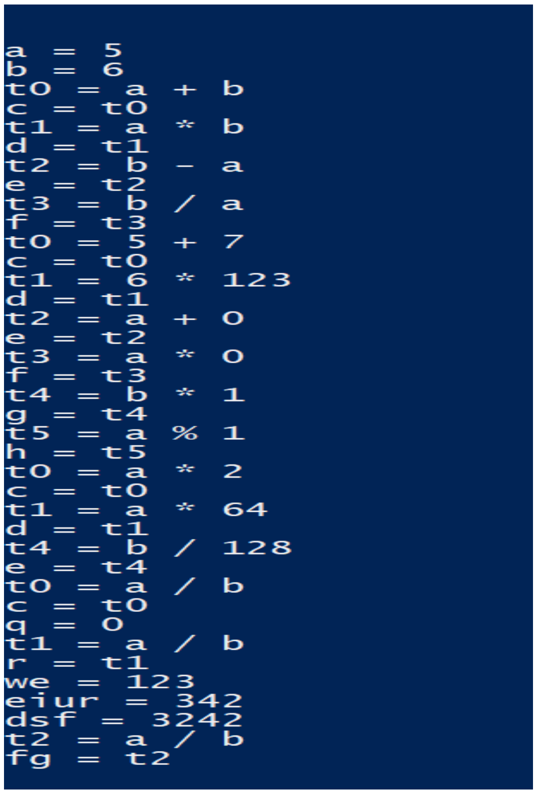
<u>Code Optimisation</u>

Sample input(Input needs to be in quadruple form)

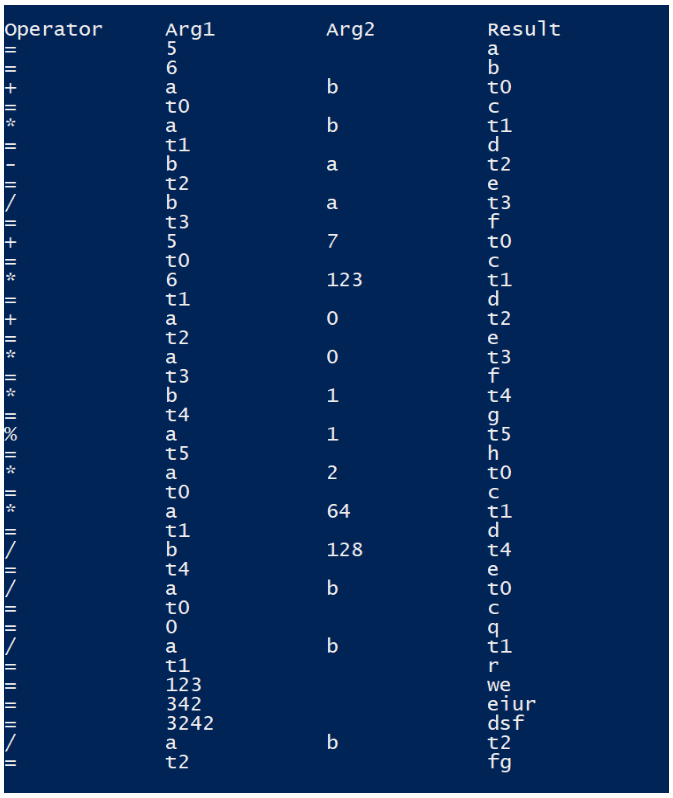
The first screenshot shows the original text file that contains the input in quadruple form.
The icg and the quadruple form of the input is printed along with the output using a function included with the functions for optimisation technique. This has been illustrated in the second screenshot.



```
=       5                a
=       6                b
+       a       b        t0
=       t0                c
*       a       b        t1
=       t1               d
-       b       a        t2
=       t2               e
/       b       a        t3
=       t3               f
+       5       7        t0
=       t0                c
*       6       123      t1
=       t1               d
+       a       0        t2
=       t2               e
*       a       0        t3
=       t3               f
*       b       1        t4
=       t4               g
%       a       1        t5
=       t5               h
*              a               2              t0
=              t0                             c
*              a               64             t1
=              t1                             d
/              b               128            t4
=              t4                             e
/       a       b        t0
=       t0               c
=       0                q
/       a       b        t1
=       t1               r
=       123              we
=       342              eiur
=       3242             dsf
/       a       b        t2
=       t2               fg
```



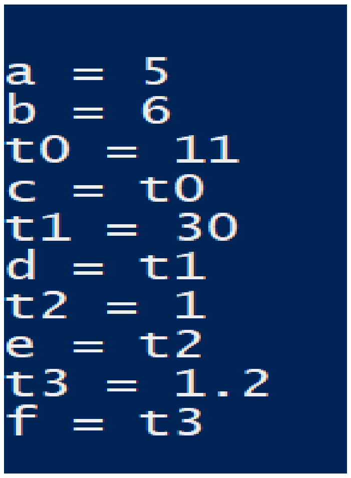
```
a = 5
b = 6
t0 = a + b
c = t0
t1 = a * b
d = t1
t2 = b - a
e = t2
t3 = b / a
f = t3
t0 = 5 + 7
c = t0
t1 = 6 * 123
d = t1
t2 = a + 0
e = t2
t3 = a * 0
f = t3
t4 = b * 1
g = t4
t5 = a % 1
h = t5
t0 = a * 2
c = t0
t1 = a * 64
d = t1
t4 = b / 128
e = t4
t0 = a / b
c = t0
q = 0
t1 = a / b
r = t1
we = 123
eiur = 342
dsf = 3242
t2 = a / b
fg = t2
```



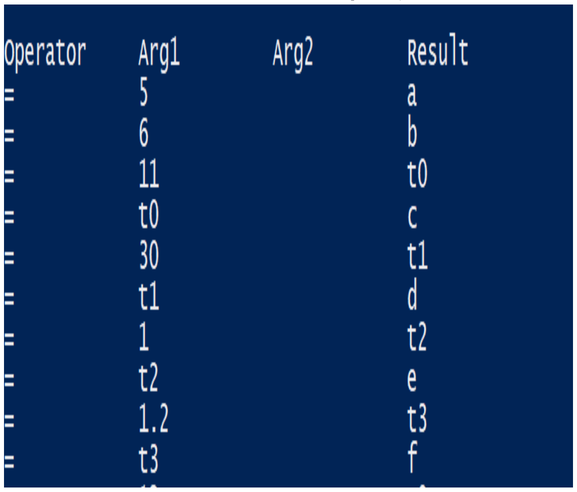
```
Operator      Arg1          Arg2          Result
=             5                           a
=             6                           b
+             a             b             t0
=             t0                          c
*             a             b             t1
=             t1                          d
-             b             a             t2
=             t2                          e
/             b             a             t3
=             t3                          f
+             5             7             t0
=             t0                          c
*             6             123           t1
=             t1                          d
+             a             0             t2
=             t2                          e
*             a             0             t3
=             t3                          f
*             b             1             t4
=             t4                          g
%             a             1             t5
=             t5                          h
*             a             2             t0
=             t0                          c
*             a             64            t1
=             t1                          d
/             b             128           t4
=             t4                          e
/             a             b             t0
=             t0                          c
=             0                           q
/             a             b             t1
=             t1                          r
=             123                         we
=             342                         eiur
=             3242                        dsf
/             a             b             t2
=             t2                          fg
```

Output

i) Constant propagation (Please refer to the first 10 lines of the input)

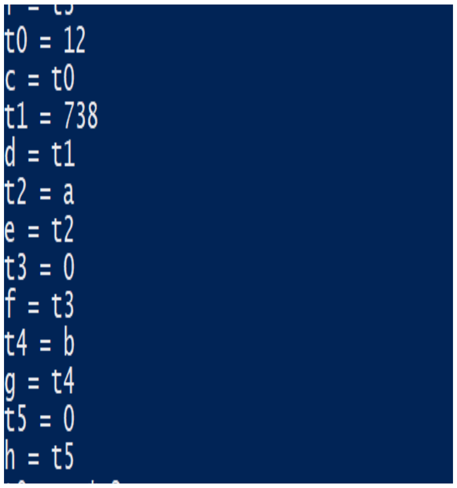


```
a = 5
b = 6
t0 = 11
c = t0
t1 = 30
d = t1
t2 = 1
e = t2
t3 = 1.2
f = t3
```

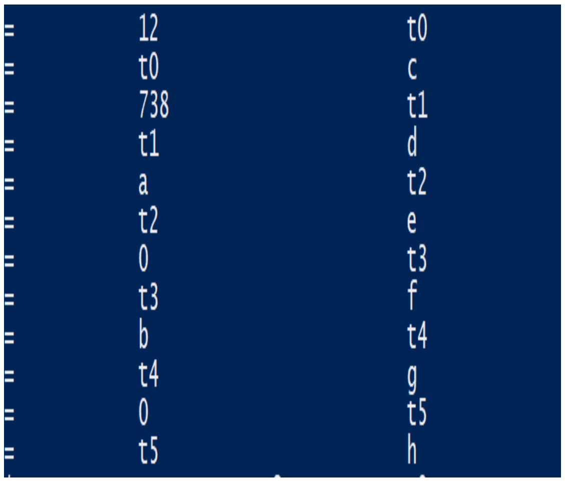


```
Operator   Arg1    Arg2    Result
=          5               a
=          6               b
=          11              t0
=          t0              c
=          30              t1
=          t1              d
=          1               t2
=          t2              e
=          1.2             t3
=          t3              f
```

ii)Constant Folding (please refer to the next 12 lines of the input)

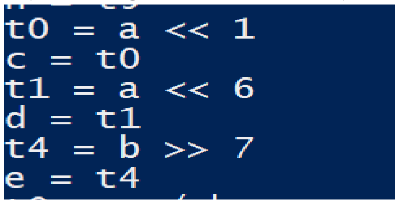


```
t0 = 12
c = t0
t1 = 738
d = t1
t2 = a
e = t2
t3 = 0
f = t3
t4 = b
g = t4
t5 = 0
h = t5
```

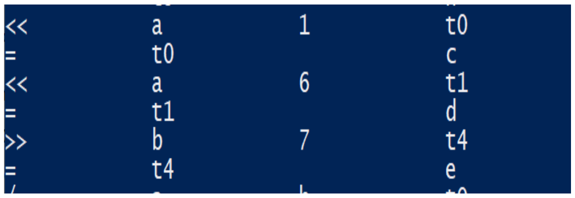


```
=        12        t0
=        t0        c
=        738       t1
=        t1        d
=        a         t2
=        t2        e
=        0         t3
=        t3        f
=        b         t4
=        t4        g
=        0         t5
=        t5        h
```

iii) Strength Reduction (Please refer to the next 6 lines of the input)

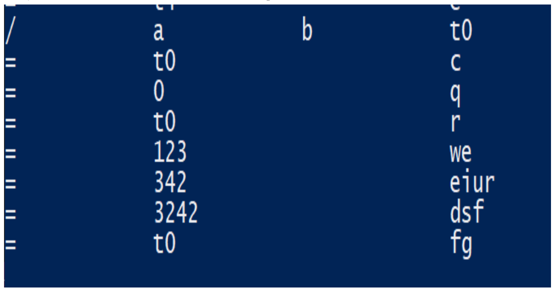


```
t0 = a << 1
c = t0
t1 = a << 6
d = t1
t4 = b >> 7
e = t4
```

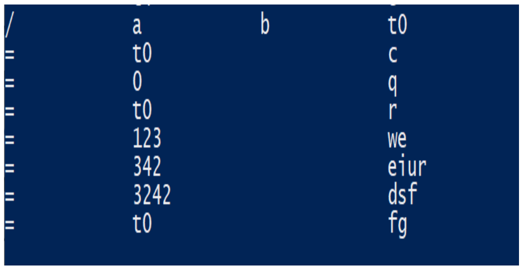


```
<<       a       1       t0
=        t0              c
<<       a       6       t1
=        t1              d
>>       b       7       t4
=        t4              e
```

iv) Common Subexpression Elimination (Please refer to the last 8 lines)



```
/        a        b        t0
=        t0                c
=        0                 q
=        t0                r
=        123               we
=        342               eiur
=        3242              dsf
=        t0                fg
```



```
/        a        b        t0
=        t0                c
=        0                 q
=        t0                r
=        123               we
=        342               eiur
=        3242              dsf
=        t0                fg
```

Conclusion

In conclusion, we learnt to build a mini-compiler, specifically for JavaScript for our project. We were able to apply all the concepts we learnt in theory classes to the project and hence got a deeper understanding of it. We familiarize ourselves with the working of Lex and Yacc tools. We understood concepts such as Context Free Grammar, Intermediate Code Generation, code optimization techniques and the like through implementation.

Further Enhancements

Further enhancements to this project can be performed. More constructs, such as for, can be included. Scripts can be written to compile function definitions, function calls and so on. More advanced techniques on error handling can be implemented as well.

References/ Bibliography

1. PESU CSE - Compiler Design Course - USE18CS351 - Class material
2. https://github.com/OmkarMetri/JavaScript-Mini-Compiler/tree/master/Tokenization - A GitHub repo
3. https://github.com/maierfelix/mini-js - A GitHub repo
4. https://github.com/RaniaBenchouiekh/mini-python-compiler - Github repo