PES UNIVERSITY

Report on

"JavaScript Mini-Compiler using Lex & Yacc"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Nitish S	**PES2201800368**
Aswin A Nair	**PES2201800504**
Sandeep Bhat	**PES2201800632**

Under the guidance of

Prof. SWATI G

Professor, Dept of CSE
PES University, Bengaluru

January – May 2021

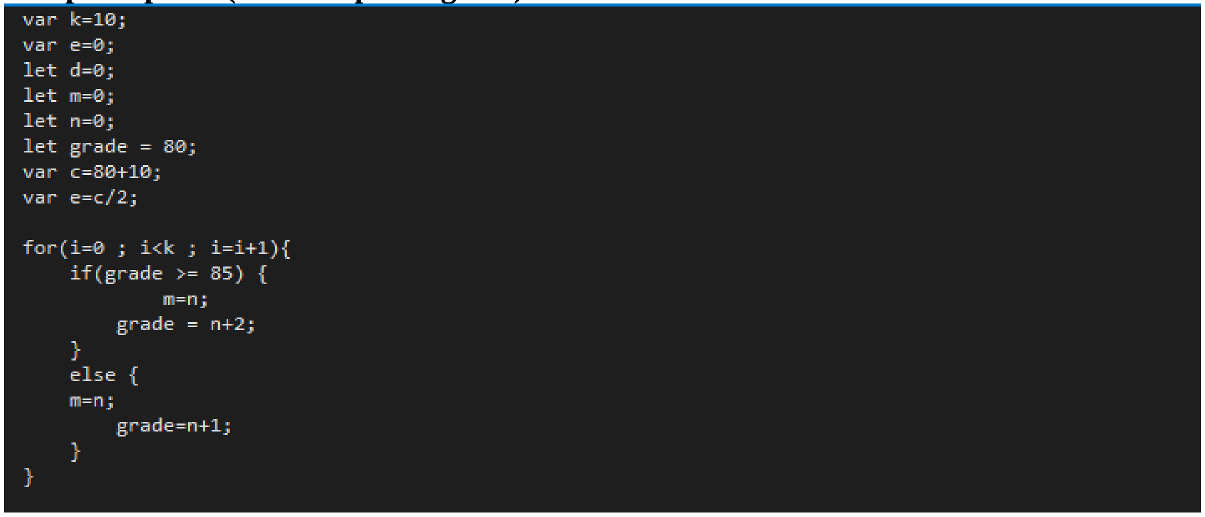
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
`
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

1. INTRODUCTION

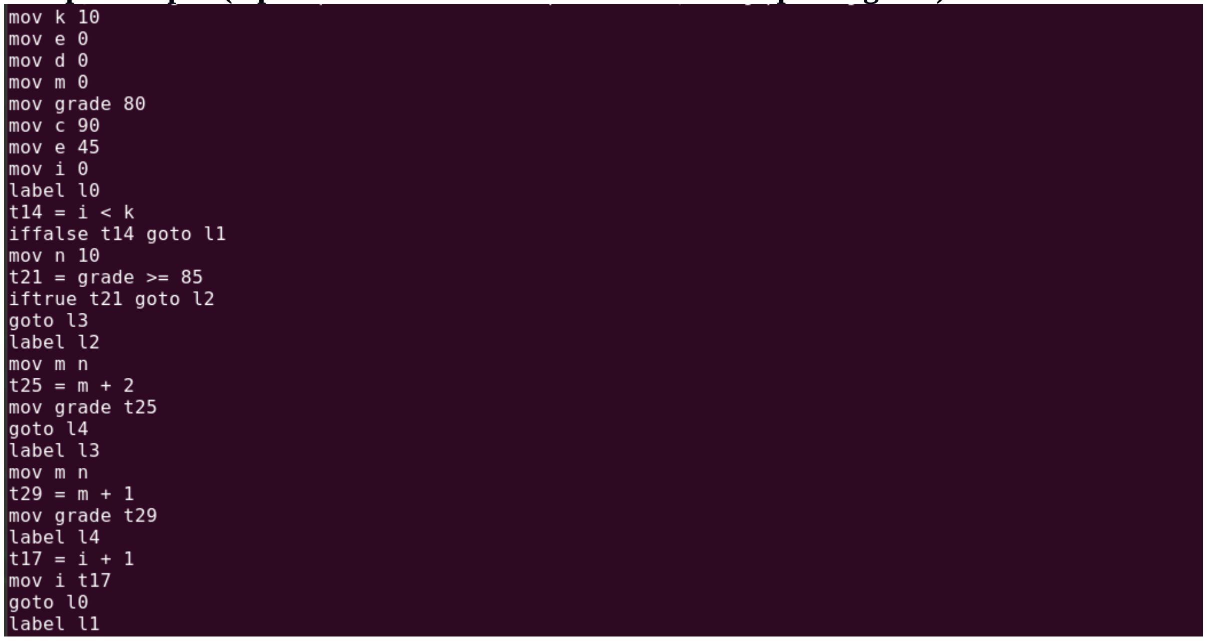
The mini compiler is built for JavaScript and handles the looping constructs namely if-else, do while and for. It is also capable of handling the basic declarations, keywords and statements like let, console.log() etc in JavaScript. The compiler is built using Lex and Yacc in combination with C language and includes Lexical Analysis, Token Generation, Syntax Analysis and Error handling, Semantic Analysis and Error handling, Abstract Syntax Tree Generation, Intermediate Code Generation and Code Optimization phases of a Compiler Design.

Sample Input: (JavaScript Program)



```
var k=10;
var e=0;
let d=0;
let m=0;
let n=0;
let grade = 80;
var c=80+10;
var e=c/2;

for(i=0 ; i<k ; i=i+1){
    if(grade >= 85) {
            m=n;
        grade = n+2;
    }
    else {
    m=n;
        grade=n+1;
    }
}
```

Sample Output (Optimized Code for the Above JavaScript Program)



```
mov k 10
mov e 0
mov d 0
mov m 0
mov grade 80
mov c 90
mov e 45
mov i 0
label l0
t14 = i < k
iffalse t14 goto l1
mov n 10
t21 = grade >= 85
iftrue t21 goto l2
goto l3
label l2
mov m n
t25 = m + 2
mov grade t25
goto l4
label l3
mov m n
t29 = m + 1
mov grade t29
label l4
t17 = i + 1
mov i t17
goto l0
label l1
```

2. ARCHITECTURE OF LANGUAGE

Our JavaScript compiler takes care of the following:

Syntax:

- Assignment Operators of String and Integer Data Types
- Arithmetic and Logical Operations
- If-Else Conditional Statement Constructs
- Do while and For loop constructs
- Single Line and Multi-Line Comments
- Declaration and print Statements

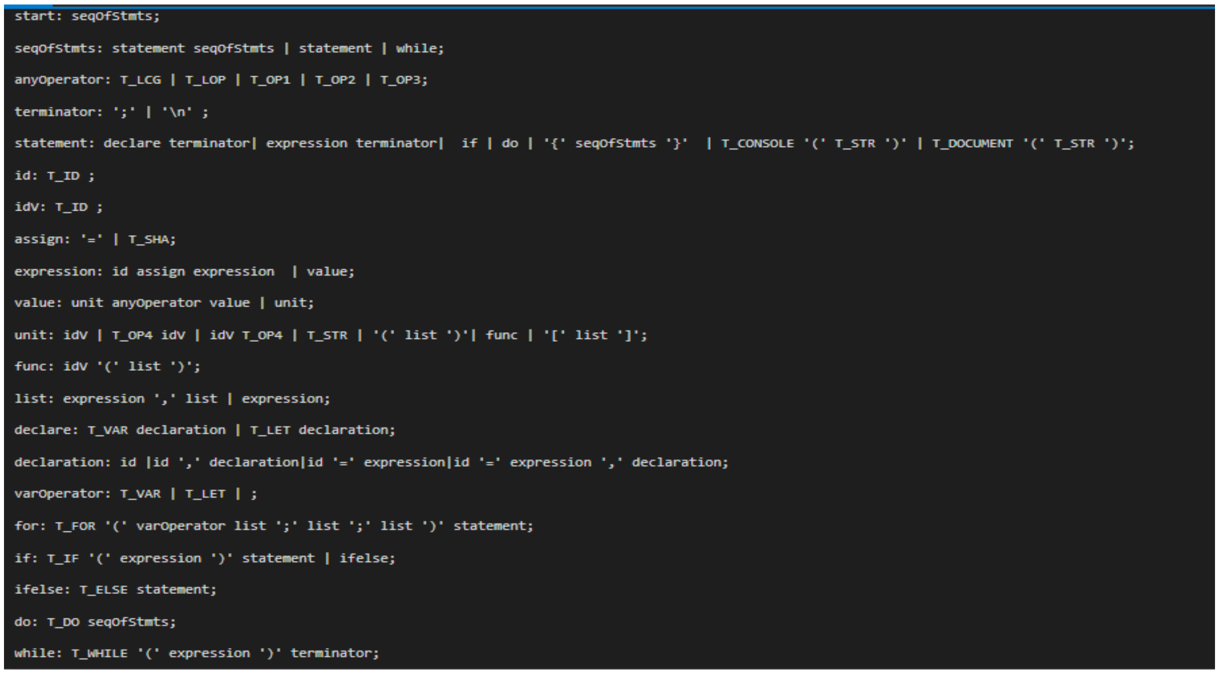
Semantics:

- We ensure that a variable is declared before its use in the program and arithmetic operations are successful only on integer values.

3. LITERATURE SURVEY/REFERENCES

- Compilers–Principles, Techniques and Tools (2nd Edition)
    - Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman
- Official Bison Documentation
- Stackoverflow

4. CONTEXT FREE GRAMMAR



```
start: seqOfStmts;

seqOfStmts: statement seqOfStmts | statement | while;

anyOperator: T_LCG | T_LOP | T_OP1 | T_OP2 | T_OP3;

terminator: ';' | '\n' ;

statement: declare terminator| expression terminator|  if | do | '{' seqOfStmts '}'  | T_CONSOLE '(' T_STR ')' | T_DOCUMENT '(' T_STR ')';

id: T_ID ;

idV: T_ID ;

assign: '=' | T_SHA;

expression: id assign expression  | value;

value: unit anyOperator value | unit;

unit: idV | T_OP4 idV | idV T_OP4 | T_STR | '(' list ')'| func | '[' list ']';

func: idV '(' list ')';

list: expression ',' list | expression;

declare: T_VAR declaration | T_LET declaration;

declaration: id |id ',' declaration|id '=' expression|id '=' expression ',' declaration;

varOperator: T_VAR | T_LET | ;

for: T_FOR '(' varOperator list ';' list ';' list ')' statement;

if: T_IF '(' expression ')' statement | ifelse;

ifelse: T_ELSE statement;

do: T_DO seqOfStmts;

while: T_WHILE '(' expression ')' terminator;
```

5. DESIGN STRATEGY

Symbol Table Generation

- We make use of a linked list of structures to implement our Symbol Table. A print function outputs a formatted symbol table to STDOUT.
- A node of the symbol table has the following structure
  - Name
  - Scope
  - Type
  - Declared Line
  - Last Used Line

Which is displayed in a tabular form during the end of the program to represent the symbol table that has been generated during the first phase.

Intermediate Code Generation

- Intermediate Code Generation is implemented using various functions and data structures that are used to generate and store the Intermediate Codes.
- We have a list of structures of type Quadruple to store the Quadruples generated by the compiler.
- Several functions are used to accomplish the required processing. We then use a print function that neatly formats the IC and prints it onto STDOUT.

Code Optimization

- Code optimization takes as an input the intermediate code which is generated and gives as an output the optimized code.
- We have performed 4 techniques of code optimization which are:
  - Constant Folding: Process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.
  - Constant Propagation: Process of substituting the values of known constants in expressions.
  - Copy Propagation: Process of replacing the occurrences of targets of direct assignments with their values.
  - Strength Reduction: Process where expensive operations are replaced with equivalent but less expensive operations.

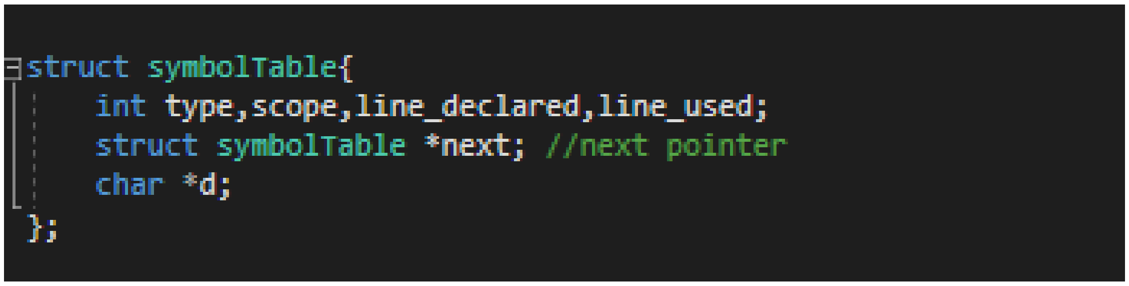
Error Handling

- We effectively handle the syntax error i.e. missing semicolon towards the end of a line of code by augmenting the grammar with an additional production rule which continues parsing by showing an appropriate error message.
- The semantic error i.e. use of a variable before declaring it, is also handled by an augmented grammar rule which continues parsing by showing an appropriate error message.

6. IMPLEMENTATION DETAILS

Symbol Table Generation

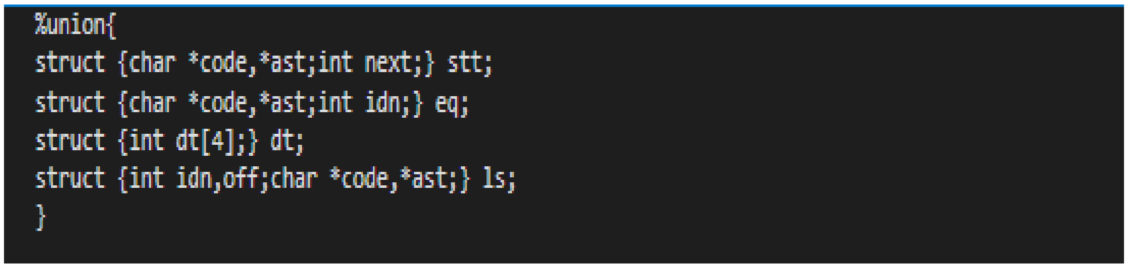
- The symbol table is constructed using a Linked List of structures as the data structure. The following structure definition is used:



```
struct symbolTable{
    int type,scope,line_declared,line_used;
    struct symbolTable *next; //next pointer
    char *d;
};
```

Intermediate Code Generation

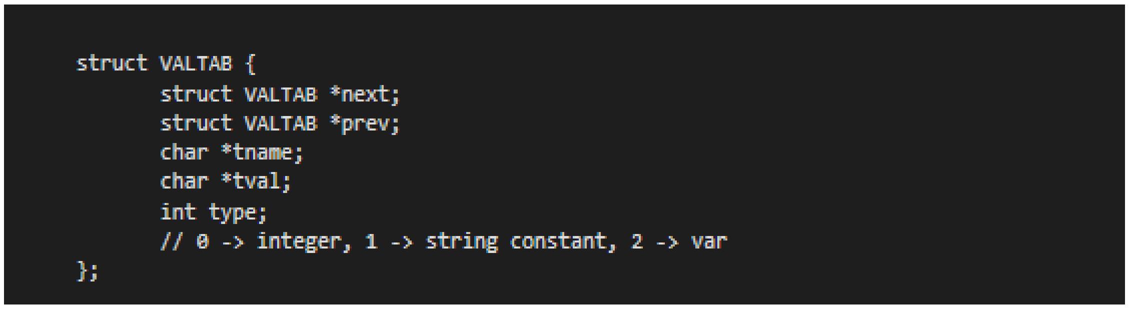
- Intermediate code was generated and stored in custom defined Quadruple structure, involving symbol table pointers and other operators.



```
%union{
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;
}
```

Code Optimization

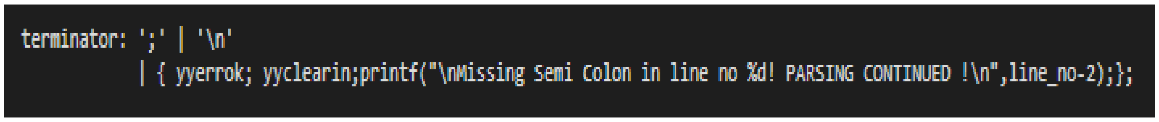
- For the code optimization phase, we implemented four methods namely: Constant Folding, Constant Propagation, Copy Propagation and Strength Reduction.
- The input was the intermediate code in the Quadruple format and we use a doubly linked list as a data structure to implement the above techniques.



```
struct VALTAB {
        struct VALTAB *next;
        struct VALTAB *prev;
        char *tname;
        char *tval;
        int type;
        // 0 -> integer, 1 -> string constant, 2 -> var
};
```

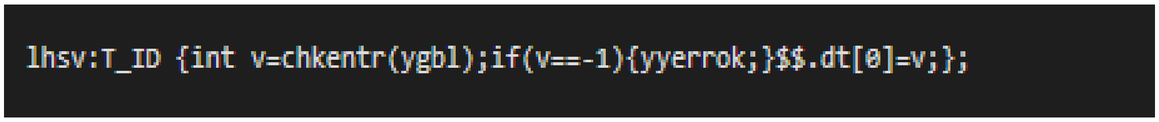
Error Handling

- We effectively handle the syntax error i.e. missing semicolon towards the end of a line of code and the semantic error i.e. using a variable before it is declared.
- Done by augmenting a grammar to the existing production rule to continue parsing by showing an appropriate error message.



```
terminator: ';' | '\n'
        | { yyerrok; yyclearin;printf("\nMissing Semi Colon in line no %d! PARSING CONTINUED !\n",line_no-2);};
```

SYNTAX ERROR HANDLING



```
lhsv:T_ID {int v=chkentr(ygbl);if(v==-1){yyerrok;}$$.dt[0]=v;};
```

SEMANTIC ERROR HANDLING

(The error message is shown by the symbol table implementation)

General instructions to run all the modules (Tokenization & Syntax Analysis, Semantic Analysis & Intermediate Code Generation and Code Optimization)

1. lex lex_filename.l
2. make -f makefile
3. ./object_filename ./input_filename.txt

Note: The paths for all the files should be given correctly for execution.

7. RESULTS AND possible shortcomings of your Mini-Compiler

The mini JavaScript compiler that we have built parses the given JavaScript grammar and as an end result generates the optimized code for any given input file.

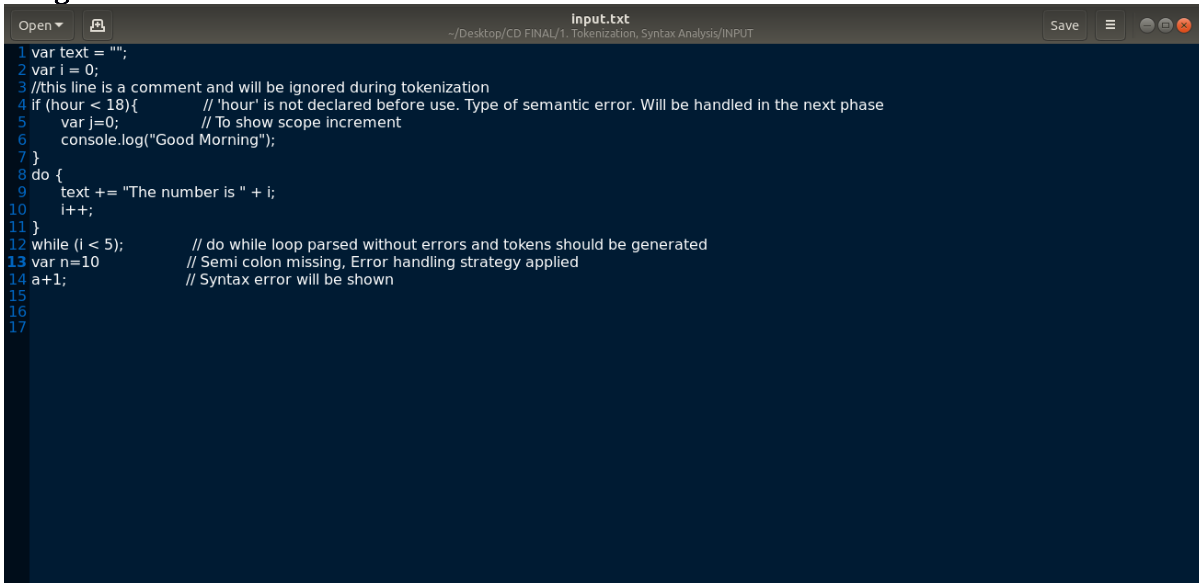
The possible shortcomings of the Mini-Compiler are:

- The compiler does not take care of Switch cases and functions as of now.
- There may be a need to run the code for optimization multiple times in order to get the final optimized code for some input files.

8. SNAPSHOTS (of different outputs)

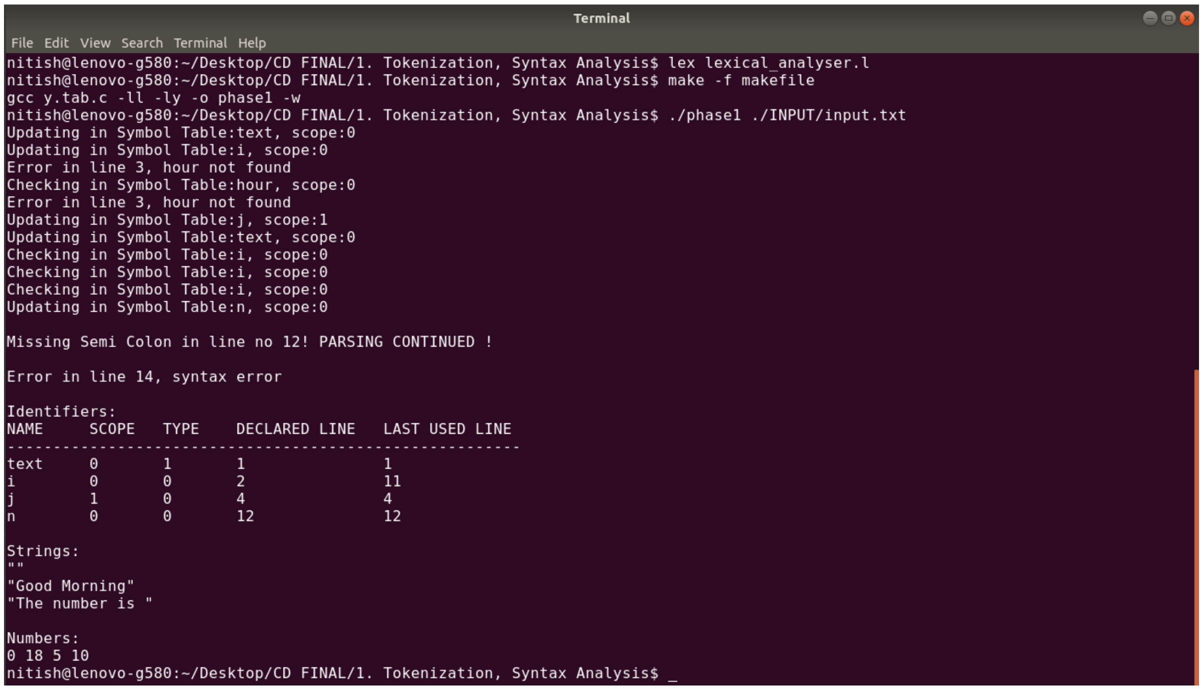
Review-1 (Tokenization, Syntax Analysis and Semantic Analysis)

Program:



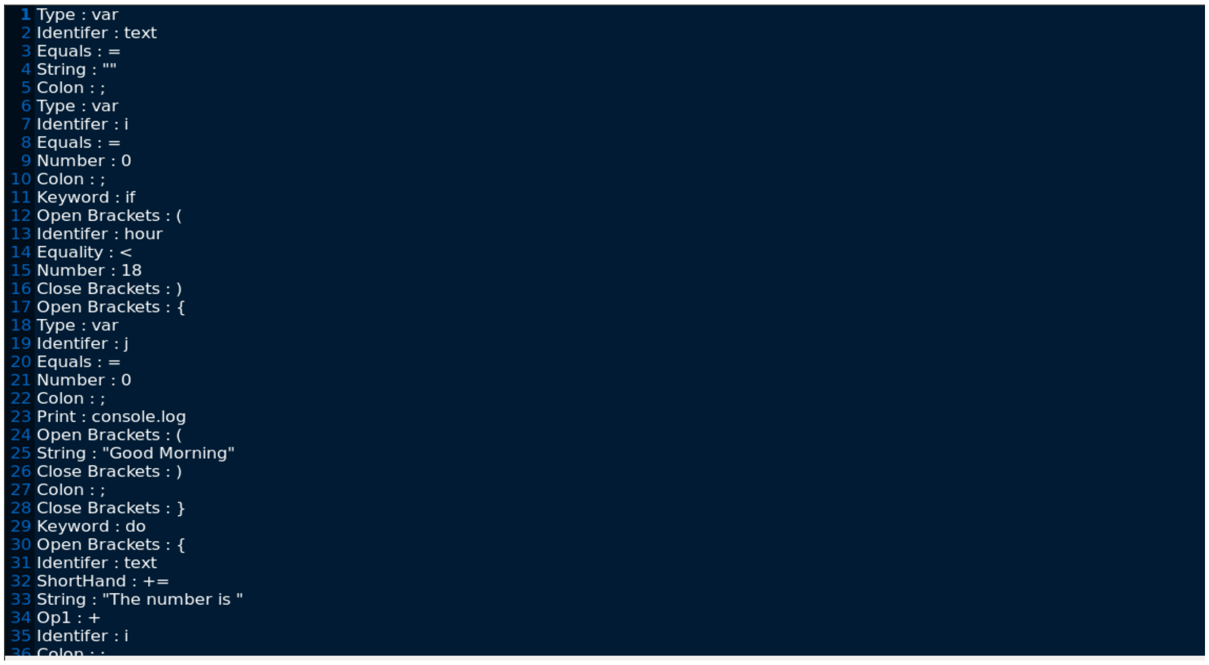
```
 1 var text = "";
 2 var i = 0;
 3 //this line is a comment and will be ignored during tokenization
 4 if (hour < 18){            // 'hour' is not declared before use. Type of semantic error. Will be handled in the next phase
 5      var j=0;              // To show scope increment
 6      console.log("Good Morning");
 7 }
 8 do {
 9      text += "The number is " + i;
10      i++;
11 }
12 while (i < 5);             // do while loop parsed without errors and tokens should be generated
13 var n=10                   // Semi colon missing, Error handling strategy applied
14 a+1;                       // Syntax error will be shown
15
16
17
```

Execution:



```
nitish@lenovo-g580:~/Desktop/CD FINAL/1. Tokenization, Syntax Analysis$ lex lexical_analyser.l
nitish@lenovo-g580:~/Desktop/CD FINAL/1. Tokenization, Syntax Analysis$ make -f makefile
gcc y.tab.c -ll -ly -o phase1 -w
nitish@lenovo-g580:~/Desktop/CD FINAL/1. Tokenization, Syntax Analysis$ ./phase1 ./INPUT/input.txt
Updating in Symbol Table:text, scope:0
Updating in Symbol Table:i, scope:0
Error in line 3, hour not found
Checking in Symbol Table:hour, scope:0
Error in line 3, hour not found
Updating in Symbol Table:j, scope:1
Updating in Symbol Table:text, scope:0
Checking in Symbol Table:i, scope:0
Checking in Symbol Table:i, scope:0
Checking in Symbol Table:i, scope:0
Updating in Symbol Table:n, scope:0

Missing Semi Colon in line no 12! PARSING CONTINUED !

Error in line 14, syntax error

Identifiers:
NAME     SCOPE   TYPE    DECLARED LINE    LAST USED LINE
---------------------------------------------------------
text     0       1       1                1
i        0       0       2                11
j        1       0       4                4
n        0       0       12               12

Strings:
""
"Good Morning"
"The number is "

Numbers:
0 18 5 10
nitish@lenovo-g580:~/Desktop/CD FINAL/1. Tokenization, Syntax Analysis$ _
```

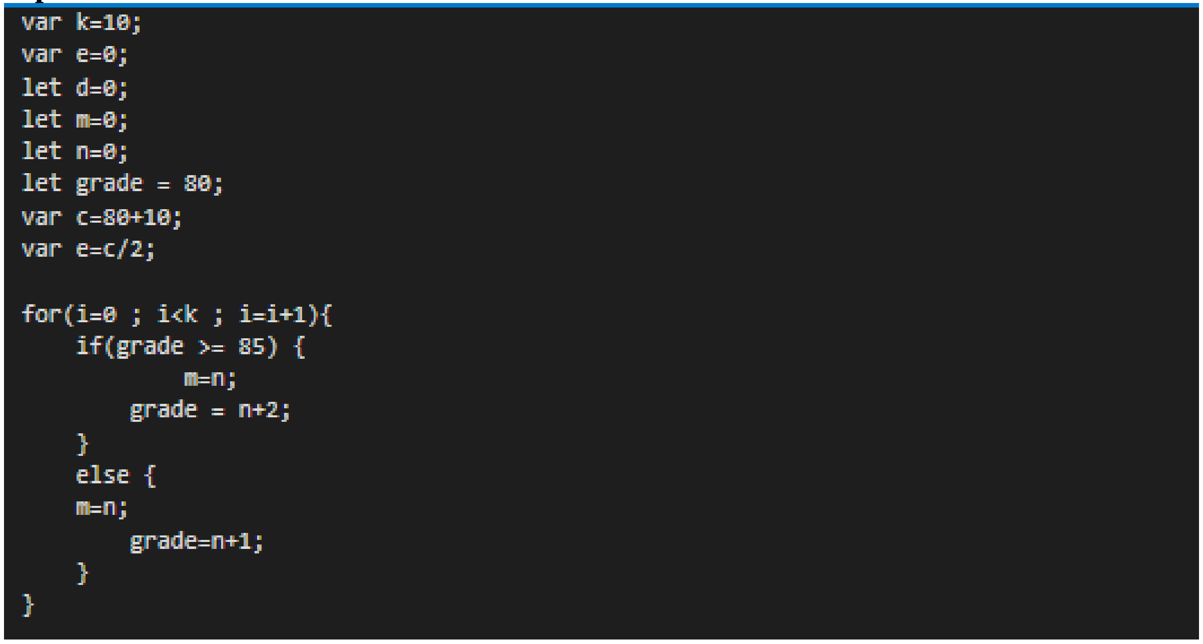
Tokens Generated:



```
Type : var
Identifer : text
Equals : =
String : ""
Colon : ;
Type : var
Identifer : i
Equals : =
Number : 0
Colon : ;
Keyword : if
Open Brackets : (
Identifer : hour
Equality : <
Number : 18
Close Brackets : )
Open Brackets : {
Type : var
Identifer : j
Equals : =
Number : 0
Colon : ;
Print : console.log
Open Brackets : (
String : "Good Morning"
Close Brackets : )
Colon : ;
Close Brackets : }
Keyword : do
Open Brackets : {
Identifer : text
ShortHand : +=
String : "The number is "
Op1 : +
Identifer : i
```

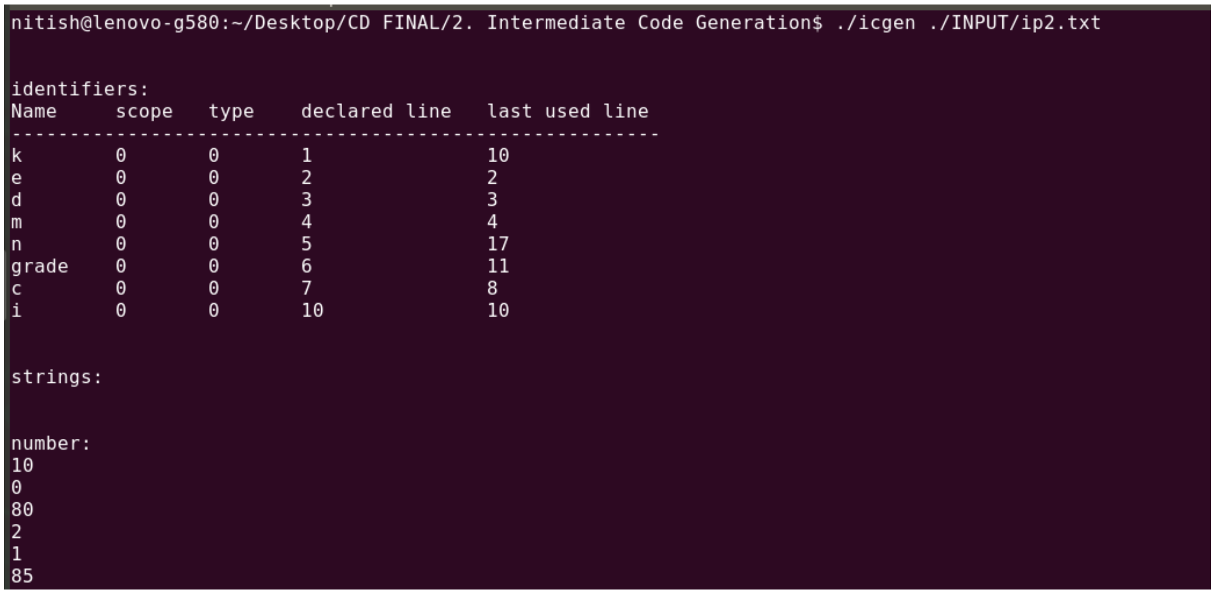
Review 2 (Intermediate Code Generation and Code Optimization)

Input File:



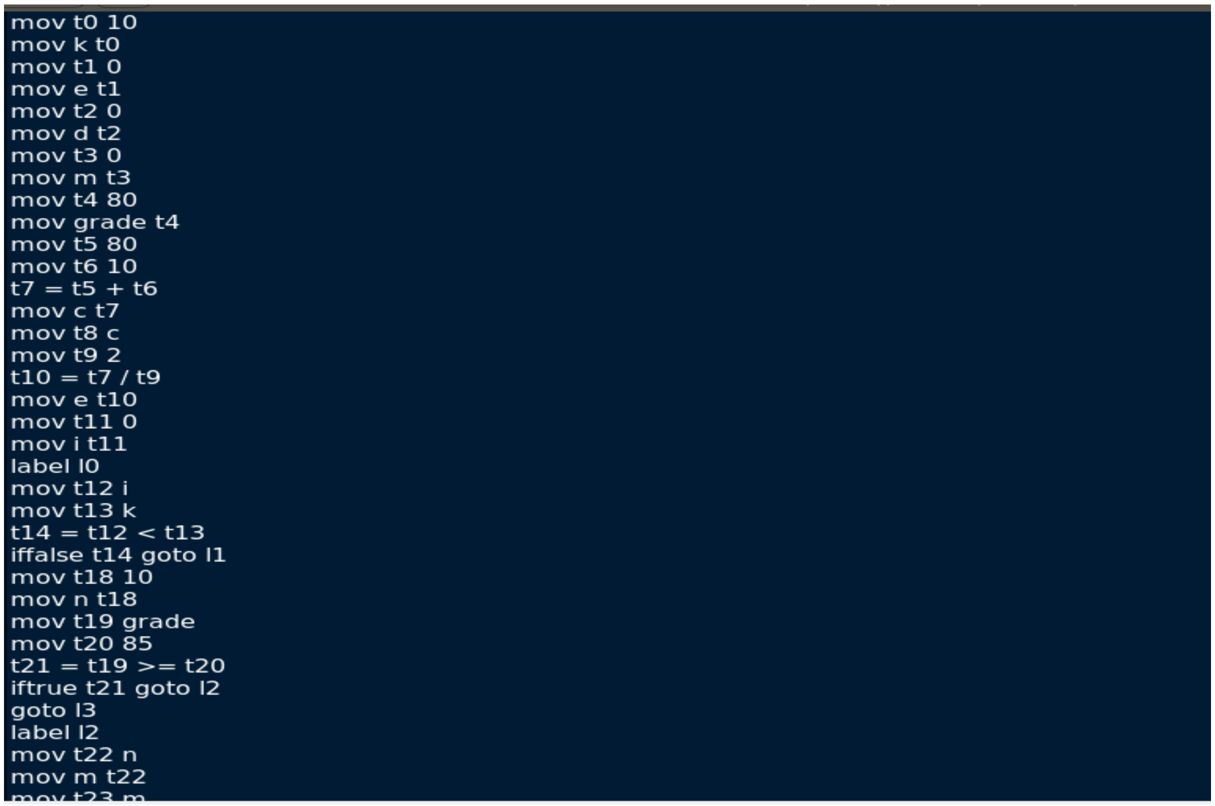
```
var k=10;
var e=0;
let d=0;
let m=0;
let n=0;
let grade = 80;
var c=80+10;
var e=c/2;

for(i=0 ; i<k ; i=i+1){
    if(grade >= 85) {
            m=n;
        grade = n+2;
    }
    else {
    m=n;
        grade=n+1;
    }
}
```

Execution:



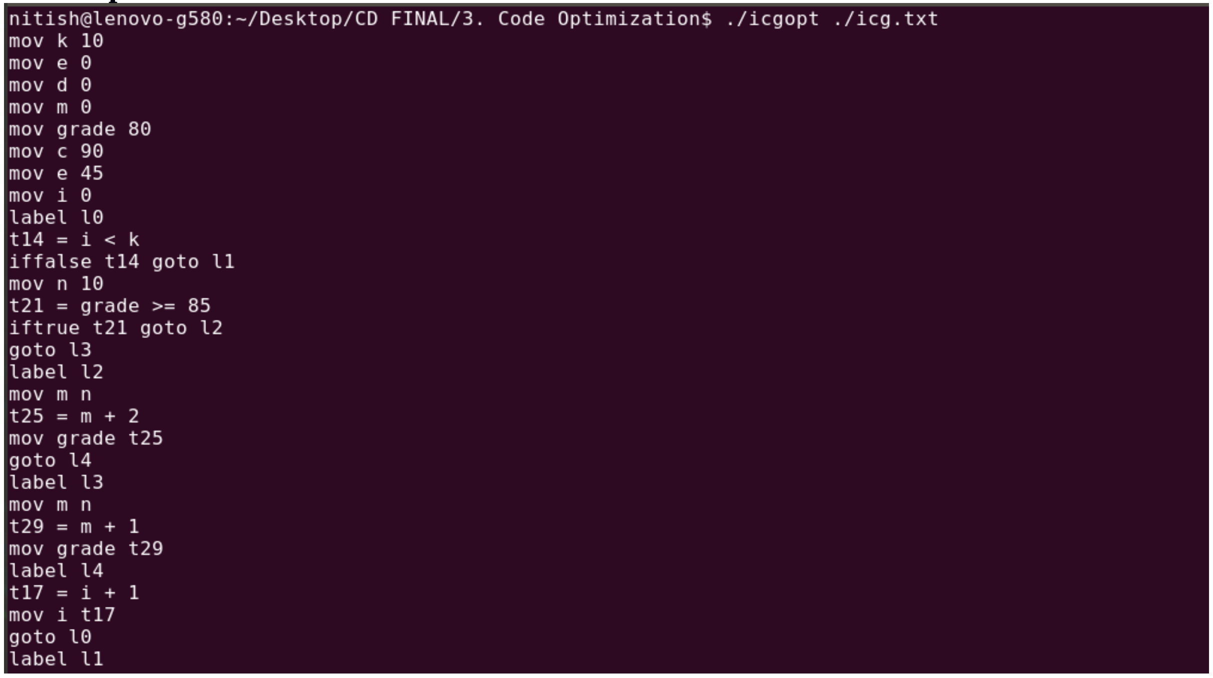
```
nitish@lenovo-g580:~/Desktop/CD FINAL/2. Intermediate Code Generation$ ./icgen ./INPUT/ip2.txt


identifiers:
Name     scope   type    declared line   last used line
--------------------------------------------------------
k        0       0       1               10
e        0       0       2               2
d        0       0       3               3
m        0       0       4               4
n        0       0       5               17
grade    0       0       6               11
c        0       0       7               8
i        0       0       10              10


strings:


number:
10
0
80
2
1
85
```

Intermediate Code Generated:



```
mov t0 10
mov k t0
mov t1 0
mov e t1
mov t2 0
mov d t2
mov t3 0
mov m t3
mov t4 80
mov grade t4
mov t5 80
mov t6 10
t7 = t5 + t6
mov c t7
mov t8 c
mov t9 2
t10 = t7 / t9
mov e t10
mov t11 0
mov i t11
label l0
mov t12 i
mov t13 k
t14 = t12 < t13
iffalse t14 goto l1
mov t18 10
mov n t18
mov t19 grade
mov t20 85
t21 = t19 >= t20
iftrue t21 goto l2
goto l3
label l2
mov t22 n
mov m t22
mov t23 m
```

Code Optimization:



```
nitish@lenovo-g580:~/Desktop/CD FINAL/3. Code Optimization$ ./icgopt ./icg.txt
mov k 10
mov e 0
mov d 0
mov m 0
mov grade 80
mov c 90
mov e 45
mov i 0
label l0
t14 = i < k
iffalse t14 goto l1
mov n 10
t21 = grade >= 85
iftrue t21 goto l2
goto l3
label l2
mov m n
t25 = m + 2
mov grade t25
goto l4
label l3
mov m n
t29 = m + 1
mov grade t29
label l4
t17 = i + 1
mov i t17
goto l0
label l1
```

9. CONCLUSIONS

A mini-compiler for JavaScript was created using Lex and Yacc with the support of C programming language. It is capable of handling the looping constructs namely if-else, do while and for. It is also capable of handling the basic declarations, keywords and statements like let, console.log() etc in JavaScript.
Basic error handling strategies have been implemented using Augmented Grammar productions for a common syntax and semantic error.

10. FURTHER ENHANCEMENTS

The created JavaScript compiler can be further enhanced by adding:

- Support for switch case and functions of JavaScript.
- More efficient optimization techniques.
- Error Recovery Strategies