Report on

“JavaScript Mini Compiler”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Sneha Jain A	**PES2201800030**
Naik Bhavan Chandrashekhar	**PES2201800047**
Varun Seshu	**PES2201800074**

Under the guidance of:
Dr. Mehala N
Professor, CSE Department
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:

This Project is a mini compiler for the JavaScript Programming Language. We have used Yacc and Lex to build .

JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.

We have implemented the for loop statement and if-else construct along with basic mathematical expression evaluation, dynamic type inference for variables and logging to the console.

The expected outcome of this project is generate a Symbol Table, an Abstract Syntax Tree and Intermediate Three-Address code along with optimization.

Sample Input:

```
1. Phase-1 > JS test.js
// testing logging numbers
console.log("***PRINTING OF NUMBERS***");
console.log(0);                  //prints 0
console.log(12);                 //positive number
console.log(132415.123456789);   //double
console.log(132415.123);         //float
console.log(-1);                 //negative
console.log(-123);
console.log(-123.123456);        //negative float

// testing 2 inline statements
console.log(1); console.log(2);
console.log(3); console.log(4)

// testing logging strings
console.log("***PRINTING OF STRINGS***");
console.log("hello")
console.log("");
console.log('hi');
console.log('');
console.log("!#@$%+-*^)");

// Print some key words
console.log("***PRINTING OF KEY-WORDS***")
console.log(true); console.log(false);
console.log(undefined)
console.log(null);

// Variable assignment
console.log("***VARIABLE ASSIGNMENT***")
var abc = 1;
var a = 2
```

Sample Output:

```
bhavan_1511@LAPTOP-AV8HL540: /mnt/c/Users/
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/U
 Phase-1$ ./jscompiler < test.js
log> ***PRINTING OF NUMBERS***
log> 0.000000000
log> 12.000000000
log> 132415.123456789
log> 132415.123000000
log> -1.000000000
log> -123.000000000
log> -123.123456000
log> 1.000000000
log> 2.000000000
log> 3.000000000
log> 4.000000000
log> ***PRINTING OF STRINGS***
log> hello
log>
log> hi
log>
log> !#@$%+-*^)
log> ***PRINTING OF KEY-WORDS***
log> true
log> false
log> undefined
log> null
log> ***VARIABLE ASSIGNMENT***
logid> abc: 1.000000
logid> abc: 12.000000
logid> a: 3.000000
logid> def: hello
```

You can find the code in our Github Repository:
https://github.com/Varun487/MiniJavaScriptCompiler

2. ARCHITECTURE OF LANGUAGE:

We have included the following functionalities of JavaScript in this mini compiler:

- Identifies comments and ignores them during execution
- Logging Numbers
- Logging Strings
- Logging 2 inline Statements
- Printing Keywords
- Variable Assignment
- Math Expression Evaluation
- Unary Expressions
- Boolean Expressions
- "FOR" Loop
- "IF" Statement

3. LITERATURE SURVEY:

- Lex & Yacc, O'Reilly, John R. Levine, Tony Mason, Doug Brown
- https://www.geeksforgeeks.org/three-address-code-compiler/
- https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm

4. CONTEXT FREE GRAMMAR:

```
line    : T_SINGLE_COMMENT end                          {;}
        | line T_SINGLE_COMMENT end                     {;}
        | T_PRINT print_exp                             {;}
        | line T_PRINT print_exp                        {;}
        | T_VAR identifier_exp end                      {;}
        | line T_VAR identifier_exp end                 {;}
        | identifier_exp end                            {;}
        | line identifier_exp end                       {;}
        | math_exp end                                  {;}
        | line math_exp end                             {;}
        | boolean_exp end                               {;}
        | line boolean_exp end                          {;}
        | T_FOR for_exp                                 {;}
        | line T_FOR for_exp                            {;}
        ;

for_exp     : T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET         T_OPEN_CURLY         line         T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET         T_OPEN_CURLY         line for_sep T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET         T_OPEN_CURLY for_sep line         T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET         T_OPEN_CURLY for_sep line for_sep T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET for_sep T_OPEN_CURLY         line         T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET for_sep T_OPEN_CURLY         line for_sep T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET for_sep T_OPEN_CURLY for_sep line         T_CLOSE_CURLY end {;}
            | T_OPEN_BRACKET for_conditions T_CLOSE_BRACKET for_sep T_OPEN_CURLY for_sep line for_sep T_CLOSE_CURLY end {;}
            ;

for_sep     : T_NEXT_LINE                               {;}
            | for_sep T_NEXT_LINE                       {;}
            ;
```

```
for_conditions  : identifier_exp T_SEMICOLON boolean_exp T_SEMICOLON identifier_exp {;}
                ;

print_exp   : T_OPEN_BRACKET print_val T_CLOSE_BRACKET end      {;}
            | T_OPEN_BRACKET T_CLOSE_BRACKET end                {printf("\n");}
            ;

print_val   : T_NUMBER                      {print_number($1);}
            | T_STRING                      {print_string($1);}
            | T_TRUE                        {printf("log> true\n");}
            | T_FALSE                       {printf("log> false\n");}
            | T_UNDEFINED                   {printf("log> undefined\n");}
            | T_NULL                        {printf("log> null\n");}
            | T_IDENTIFIER                  {print_identifier($1);}
            | math_exp                      {print_number($1);}
            ;
```

```
boolean_exp     : T_NUMBER T_EQ T_NUMBER            {$$ = eq($1, $3);}
                | T_IDENTIFIER T_EQ T_NUMBER        {$$ = eq(get_id_num($1), $3);}
                | T_NUMBER T_EQ T_IDENTIFIER        {$$ = eq($1, get_id_num($3));}
                | T_IDENTIFIER T_EQ T_IDENTIFIER    {$$ = eq(get_id_num($1), get_id_num($3));}
                | T_NUMBER T_NEQ T_NUMBER           {$$ = neq($1, $3);}
                | T_IDENTIFIER T_NEQ T_NUMBER       {$$ = neq(get_id_num($1), $3);}
                | T_NUMBER T_NEQ T_IDENTIFIER       {$$ = neq($1, get_id_num($3));}
                | T_IDENTIFIER T_NEQ T_IDENTIFIER   {$$ = neq(get_id_num($1), get_id_num($3));}
                | T_NUMBER T_LT T_NUMBER            {$$ = lt($1, $3);}
                | T_IDENTIFIER T_LT T_NUMBER        {$$ = lt(get_id_num($1), $3);}
                | T_NUMBER T_LT T_IDENTIFIER        {$$ = lt($1, get_id_num($3));}
                | T_IDENTIFIER T_LT T_IDENTIFIER    {$$ = lt(get_id_num($1), get_id_num($3));}
                | T_NUMBER T_LTE T_NUMBER           {$$ = lte($1, $3);}
                | T_IDENTIFIER T_LTE T_NUMBER       {$$ = lte(get_id_num($1), $3);}
                | T_NUMBER T_LTE T_IDENTIFIER       {$$ = lte($1, get_id_num($3));}
                | T_IDENTIFIER T_LTE T_IDENTIFIER   {$$ = lte(get_id_num($1), get_id_num($3));}
                | T_NUMBER T_GT T_NUMBER            {$$ = gt($1, $3);}
                | T_IDENTIFIER T_GT T_NUMBER        {$$ = gt(get_id_num($1), $3);}
                | T_NUMBER T_GT T_IDENTIFIER        {$$ = gt($1, get_id_num($3));}
                | T_IDENTIFIER T_GT T_IDENTIFIER    {$$ = gt(get_id_num($1), get_id_num($3));}
                | T_NUMBER T_GTE T_NUMBER           {$$ = gte($1, $3);}
                | T_IDENTIFIER T_GTE T_NUMBER       {$$ = gte(get_id_num($1), $3);}
                | T_NUMBER T_GTE T_IDENTIFIER       {$$ = gte($1, get_id_num($3));}
                | T_IDENTIFIER T_GTE T_IDENTIFIER   {$$ = gte(get_id_num($1), get_id_num($3));}
                ;
```

```
identifier_exp  : T_IDENTIFIER T_ASSIGN T_NUMBER        {update_symbol_table_number($1, $3);}
                | T_IDENTIFIER T_ASSIGN T_STRING        {update_symbol_table_string($1, $3);}
                | T_IDENTIFIER T_ASSIGN T_TRUE          {update_symbol_table_bool($1, 2);}
                | T_IDENTIFIER T_ASSIGN T_FALSE         {update_symbol_table_bool($1, 3);}
                | T_IDENTIFIER T_ASSIGN T_NULL          {update_symbol_table_bool($1, 4);}
                | T_IDENTIFIER T_ASSIGN T_UNDEFINED     {update_symbol_table_bool($1, 5);}
                | T_IDENTIFIER T_ASSIGN math_exp        {update_symbol_table_number($1, $3);}
                | T_IDENTIFIER T_INCREMENT              {increment($1);}
                | T_IDENTIFIER T_DECREMENT              {decrement($1);}
                | T_INCREMENT T_IDENTIFIER              {increment($2);}
                | T_DECREMENT T_IDENTIFIER              {decrement($2);}
                | T_IDENTIFIER T_INC_ASSIGN T_NUMBER    {inc_assign($1, $3);}
                | T_IDENTIFIER T_DEC_ASSIGN T_NUMBER    {dec_assign($1, $3);}
                | T_IDENTIFIER T_MUL_ASSIGN T_NUMBER    {mul_assign($1, $3);}
                | T_IDENTIFIER T_DIV_ASSIGN T_NUMBER    {div_assign($1, $3);}
                | T_IDENTIFIER T_REM_ASSIGN T_NUMBER    {rem_assign($1, $3);}
                ;

math_exp    : T_NUMBER '+' T_NUMBER                 {$$ = $1 + $3;}
            | T_NUMBER '-' T_NUMBER                 {$$ = $1 - $3;}
            | T_NUMBER '*' T_NUMBER                 {$$ = $1 * $3;}
            | T_NUMBER '/' T_NUMBER                 {$$ = $1 / $3;}
            | T_NUMBER '%' T_NUMBER                 {$$ = rem($1, $3);}
            | math_exp '+' T_NUMBER                 {$$ = $1 + $3;}
            | math_exp '-' T_NUMBER                 {$$ = $1 - $3;}
            | math_exp '*' T_NUMBER                 {$$ = $1 * $3;}
            | math_exp '/' T_NUMBER                 {$$ = $1 / $3;}
            | math_exp '%' T_NUMBER                 {$$ = rem($1, $3);}
            ;

end     : T_NEXT_LINE                       {;}
        | T_NEXT_LINE end                   {;}
        | T_SEMICOLON                       {;}
        | T_SEMICOLON end                   {;}
        ;
```

5. DESIGN STRATEGY:

- **SYMBOL TABLE CREATION:**
  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.

  Our design of a symbol table consisted of an array of structure pointers in C which pointed to structures which contained all the information regarding a particular variable, constants, temporary variables and labels being stored. So in effect we have an array of structure pointers, where each structure stores a unique id per variable, a data type flag to mention the datatype of the variable, a scope flag which mentions the scope of the variables, an occupied flag to determine whether or not the variable has a value or is just declared and a union to store the data that the variable might hold.

- **INTERMEDIATE CODE GENERATION:**

  In our design, the semantic actions that are executed while parsing the JavaScript program are responsible for intermediate code generation.

  For label generation, we have implemented a counter to generate unique labels for each use case (the for and if constructs). Similarly, there is a global temporary variable counter which helps generate unique temporary variables for use cases such as assignments, comparisons, for loop, etc.

- **CODE OPTIMIZATION:**

  - **Constant Folding:** The process of recognizing and evaluating constant expressions at compile time instead of run time is called constant folding.

  - **Constant propagation:** The process of substituting the values of known constants in expressions. Constants used in an expression are combined, and new ones are generated. Some implicit conversions between integers and floating-point types are done.

  - **Dead Code Elimination:** It removes code which does not affect the program's execution. It helps shrink program size and decreases the program's running time.

  - **Common Subexpression Elimination:** It searches for instances of identical expressions and analyzes whether it is worthwhile to replace them with a single variable holding the computed value.

- **ERROR HANDLING:**

  - We are handling syntax errors, which are generated during parsing.
  - When a lexical error is encountered, the program execution is stopped and the line number of the error is mentioned.
  - We are also handling re-declaration of variables in the same scope, and are showing appropriate error messages.
  - Upon encountering an error, our compiler continues parsing through the rest of the code and lists all errors encountered during parsing with the line number.

6. IMPLEMENTATION DETAILS:

- **SYMBOL TABLE CREATION:**

Data Structure

```
// struct for identifer
struct identifier{
    char *id;
    int datatype_flag;
    int occupied;
    union data {
        char *str;
        double num;
    } data;
} identifier;
```

Symbol table functions implemented

```
// Makes identifier string
char *convert_identifer(char *identifer);

// get the index of an existing entry or to add a new entry
int compute_symbol_index(char *symbol);

// Updates symbol table with a variable of number type
void update_symbol_table_number(char *symbol, double val);

// Updates symbol table with a variable of string type
void update_symbol_table_string(char *symbol, char *val);

// Updates symbol table with a variable of boolean type
void update_symbol_table_bool(char *symbol, int type);

// Given symbol character, will look up value of identifier in symbol table
// returns a void pointer to the value of the symbol and
// sets the type value to the datatype of the value
void *get_symbol_value(char *symbol, int *type);

// print the value of an identifier
void print_identifier(char *symbol);

// Given symbol character, will look up value int of that in symbol table
int  getSymbolVal(char symbol);

// Given symbol and value, will make sure that the respective symbol gets that value
void updateSymbolVal(char symbol, int val);
```

- **INTERMEDIATE CODE GENERATION:**

Writing the intermediate code to a file and generation of intermediate code for a statement

```
start:seq{FILE *f;f=fopen("../icg.txt","w");
    printf("\n\nGiven Code:\n\n%s\n\nThree Address Code (TAC):\n\n%s",$1.ast,$1.code);
    fprintf(f,"%s",$1.code);
    fclose(f);};
estt:{;};
elb:{;};
edt:{;};

anyopl:T_LOP {$$.code=strdup(ysign);}|T_OP1 {$$.code=strdup(ysign);}|T_LCG {$$.code=strdup(ysign);};

anyoph:T_OP2 {$$.code=strdup(ysign);}|T_OP3 {$$.code=strdup(ysign);};

adlm:';'|'\n';

seq:statement seq{$$.code=ap($1.code,$2.code);$$.ast=ap3($1.ast,strdup("\n"),$2.ast);}|
    {$$.code=strdup("");$$.ast=strdup("");};

statement:defn adlm {$$.code=$1.code;$$.ast=$1.ast;}|
    expr adlm {$$.code=$1.code;$$.ast=$1.ast;}|
    for {$$.code=$1.code;$$.ast=$1.ast;}|
    if {$$.code=$1.code;$$.ast=$1.ast;}|
    '{'{scs[stop++]=sid++;} seq '}' {stop--;} adlm
    {$$.code=$3.code;$$.ast=ap3(strdup("{"),$3.ast,strdup("} "));}|
    adlm {$$.code=strdup("");$$.ast=strdup("");};
```

Intermediate code generated - for and if constructs

```
for: T_FOR edt {$2.dt[0]=lbl++;$2.dt[1]=lbl++;}
    '(' expr ';' expr ';' expr ')' statement {char *a,*b,*c;
    sprintf(bbuf,"label l%d :\n",$2.dt[0]);
    a=ap3($5.code,strdup(bbuf),$7.code);
    sprintf(bbuf,"iffalse t%d goto l%d\n",$7.idn,$2.dt[1]);
    b=ap3(strdup(bbuf),$11.code,$9.code);
    sprintf(bbuf,"goto l%d\nlabel l%d :\n",$2.dt[0],$2.dt[1]);
    $$.code=ap3(a,b,strdup(bbuf));
    a=ap3(strdup("for ("),$5.ast,strdup(";"));
    a=ap3(a,$7.ast,strdup(";"));
    a=ap3(a,$9.ast,strdup(")"));
    $$.ast=ap(a,$11.ast);
};

if : T_IF '('expr')' edt
    {$5.dt[0]=lbl++;$5.dt[1]=lbl++;$5.dt[2]=lbl++;}
    statement T_ELSE statement
    {char *a,*b,*c;
    sprintf(bbuf,"iftrue t%d goto l%d\ngoto l%d\nlabel l%d :\n",$3.idn,$5.dt[0],$5.dt[1],$5.dt[0]);
    a=ap3($3.code,strdup(bbuf),$7.code);
    sprintf(bbuf,"goto l%d\nlabel l%d :\n",$5.dt[2],$5.dt[1]);
    b=ap(strdup(bbuf),$9.code);
    sprintf(bbuf,"label l%d :\n",$5.dt[2]);
    $$.code=ap3(a,b,strdup(bbuf));
    a=ap3(strdup("if("),$3.ast,strdup(")"));
    a=ap3(a,$7.ast,strdup(" else "));
    $$.ast=ap(a,$9.ast);
```

- **CODE OPTIMIZATION:**

Optimization of Intermediate 3 address code generated

```
statement: T_TVAR T_ASSIGN value_prod
            | T_AVAR T_ASSIGN value_constants {check_update(&$1.value, &$2.value, &$3.value);printf("%s = %s\n", $1.
            | T_TVAR T_ASSIGN T_NUM operator no_num {print_current_vt();check_update(&$1.value, &$3.value, &$5.value
            | T_TVAR T_ASSIGN no_num operator T_NUM {print_current_vt();check_update(&$1.value, &$3.value, &$5.value
            | T_TVAR T_ASSIGN T_NUM operator T_NUM {printf("mov %s %s\n", $1.value, constant_fold($3.value, $4.value
            | T_TVAR T_ASSIGN no_num operator no_num {print_current_vt();check_update(&$1.value, &$3.value, &$5.valu
            | T_FALSE value_constants T_GOTO T_LVAR T_DELIM{check_update(&$1.value, &$2.value, &$3.value);printf("%s
            | T_TRUE value_constants T_GOTO T_LVAR T_DELIM{check_update(&$1.value, &$2.value, &$3.value);printf("%s
            | T_GOTO T_LVAR  {check_update(&$1.value, &$2.value, &$2.value);printf("%s %s\n", $1.value, $2.value);}
            | T_LABEL T_LVAR T_COLON{check_update(&$1.value, &$2.value, &$2.value);printf("%s %s :\n", $1.value, $2.
            | T_TVALVEC T_ASSIGN T_TVAR {check_update(&$1.value, &$2.value, &$3.value);printf("%s %s %s\n", $1.value
            | T_TVAR '[' tvar_tnum ']' T_ASSIGN tvar_tnum {check_update(&$1.value, &$3.value, &$6.value);printf("%s[
            | T_TVAR T_ASSIGN value_constants '[' value_constants ']' {check_update(&$1.value, &$3.value, &$5.value)
```

One of the Techniques (Constant Folding)

```
char* constant_fold(char *operand1, char *operator, char *operand2)
{
    int op1 = atoi(operand1), op2 = atoi(operand2);
    int result;
    char *returnval = (char*)malloc(sizeof(char)*32);

    if(strcmp(operator, "&&") == 0){
        result = op1 && op2;
        sprintf(returnval, "%d", result);
        return(returnval);
    }

    if(strcmp(operator, "||") == 0){
        result = op1 || op2;
        sprintf(returnval, "%d", result);
        return(returnval);
    }

    if(strcmp(operator, "<") == 0){
        result = op1 < op2;
        sprintf(returnval, "%d", result);
        return(returnval);
    }

    if(strcmp(operator, ">") == 0){
        result = op1 > op2;
        sprintf(returnval, "%d", result);
        return(returnval);
    }

    if(strcmp(operator, "<=") == 0){
        result = op1 <= op2;
        sprintf(returnval, "%d", result);
        return(returnval);
    }
```

- **ERROR HANDLING:**

  Handling undeclared variables

```
void yyerror(char *a){printf("\nError in line %d, %s\n", elno,a);return;}
int main(){
    yyparse();
    //printall();
    return 0;
}
```

  Lexical Error Handling

```
.                    {ECHO; yyerror ("Lexical error: unexpected character");}

%%
```

7. RESULTS:

We have been able to successfully build a mini-compiler for JavaScript and implement all the functionalities mentioned in the Architecture of the Language (Section 2 of this Document). We have created a CFG to parse code in JavaScript. Further, we generated intermediate code in the 3 address code format and applied code optimization concepts on our generated code. Thus, we have implemented a compiler which takes a JavaScript program as input and generates an optimized 3 address code as the output.

8. SNAPSHOTS:

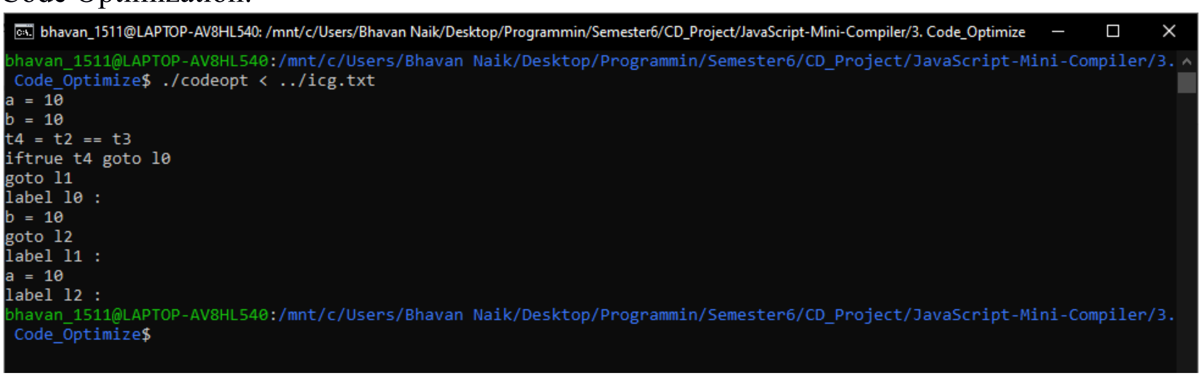
Working of Basic Operations:

```
bhavan_1511@LAPTOP-AV8HL540: /mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/1. Phase-1
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/1.
 Phase-1$ ./jscompiler < test.js
log> ***PRINTING OF NUMBERS***
log> 0.000000000
log> 12.000000000
log> 132415.123456789
log> 132415.123000000
log> -1.000000000
log> -123.000000000
log> -123.123456000
log> 1.000000000
log> 2.000000000
log> 3.000000000
log> 4.000000000
log> ***PRINTING OF STRINGS***
log> hello
log>
log> hi
log>
log> !#@$%+-*^)
log> ***PRINTING OF KEY-WORDS***
log> true
log> false
log> undefined
log> null
log> ***VARIABLE ASSIGNMENT***
logid> abc: 1.000000
logid> abc: 12.000000
logid> a: 3.000000
logid> def: hello
```

Intermediate Code Generation:

```
bhavan_1511@LAPTOP-AV8HL540: /mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/2. ICG
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/2. ICG$ ./icg < inputFiles/ip1.txt

Given Code:

a = (10)

b = (15)

if( (a) == (10) ){
b = (100)

}  else {
a = (100)

}

Three Address Code (TAC):

t0 = 10
a = t0
t1 = 15
b = t1
t2 = a
t3 = 10
t4 = t2 == t3
iftrue t4 goto l0
goto l1
label l0 :
t5 = 100
b = t5
goto l2
label l1 :
t6 = 100
a = t6
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/2. ICG$
```

Code Optimization:



```
bhavan_1511@LAPTOP-AV8HL540: /mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/3. Code_Optimize
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/3.
 Code_Optimize$ ./codeopt < ../icg.txt
a = 10
b = 10
t4 = t2 == t3
iftrue t4 goto l0
goto l1
label l0 :
b = 10
goto l2
label l1 :
a = 10
label l2 :
bhavan_1511@LAPTOP-AV8HL540:/mnt/c/Users/Bhavan Naik/Desktop/Programmin/Semester6/CD_Project/JavaScript-Mini-Compiler/3.
 Code_Optimize$
```

9. CONCLUSIONS:

We have built a mini-compiler of JavaScript using lex and yacc implementing functions like console.log(), for-loop and if statements (a full list of all functions implemented is present in section 2). By doing this project, we have gained a better insight into the phases of the compiler. We created a Context Free Grammar (CFG) for the JavaScript language. Next, we generated intermediate code in a Three Address Code Format. We finally optimize the Three address code using codes and display the results. Further, we have implemented error handling for lexical and syntax errors.

10. FURTHER ENHANCEMENTS:

We can extend this project further and make it a full-fledged JavaScript Compiler including extensive error documentation and correction, implementing while, for-each loop, functions, etc. We could also incorporate more code optimization techniques such as copy propagation, strength reduction, loop invariant variable detection, etc.

REFERENCES:

- Lex & Yacc, O'Reilly, John R. Levine, Tony Mason, Doug Brown
- https://www.geeksforgeeks.org/code-optimization-in-compiler-design/
- https://github.com/r-i-c-h-a/R-Mini-Compiler
- https://youtu.be/54bo1qaHAfk
- https://youtu.be/__-wUHG2rfM
- Our Github Repository: https://github.com/Varun487/MiniJavaScriptCompiler