COMPILER ASSIGNMENT

Language Choosen:- Objective C

Constructs Used:-

- **Keywords**: int, float, string, while, if, else, true, false
- Operators: +, -, *, /, %, =, ==, >, <, >=, <=, !=, &&, ||, !
- Delimiters: {, }, (,), [,], ;, ,
- **Identifiers**: must start with a letter (upper or lower case), and may contain zero or more additional characters as long as they are letters, digits, or underscores
- **Integer Literals**: may begin with an optional plus or minus followed by a sequence of one or more digits, provided that the first digit can only be zero for the number zero (which should not have a plus or minus before it).
- **Floating Point Literals**: may begin with an optional plus or minus followed by a sequence of one or more digits with the same provision above for integers, followed by a decimal point and one or more digits after the decimal point.
- **String literals**: In objective C data type used for strings is **NSString** and the format goes like this:- **NSString *id=@"string literal";** here instead of making ***** as an overloaded operator we used **#** so we do not get conflicts with multiply operator(*) while parsing.

Grammar Used:-

S-> int main ( ) { STMTS return 0 ; }

STMTS-> STMT R

R-> STMTS | null

STMT-> IF | WHILE | ASS | DEC

DEC-> TYPE LIST ;

LIST-> id LIST1

LIST1-> , LIST | null

IF-> if (E) { STMTS } IF1

IF1-> else{ STMTS } | null

WHILE -> while(E) { STMTS }

ASS -> id = E;

DEC1-> TYPE LIST2 ;

LIST2 -> #id LIST3

LIST3 -> , LIST2 | null

TYPE-> int | float | char | double | long | NSString | bool

Grammar for expression:-

E -> T

T-> F T1

T1-> || F T1

T1-> null

F -> A F1

F1 -> && A F1 | null

A -> B A1

A1 -> X A1 | null

X -> == B | != B

B -> C B1

B1 -> K B1| null

K-> < C | > C | <= C | >= C

C-> D C1

C1 -> N C1 | null

N -> + D | - D

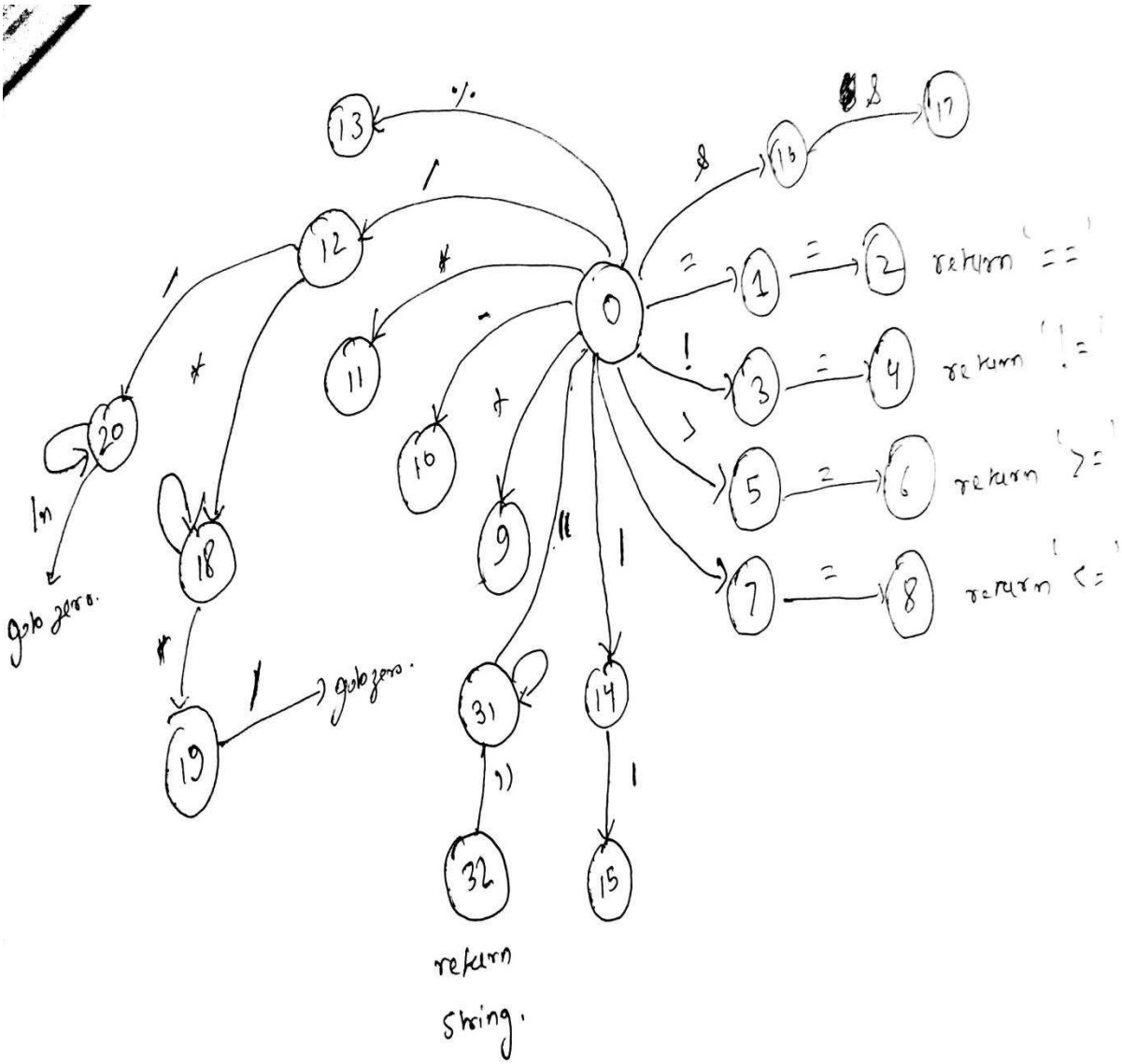
B -> G D1

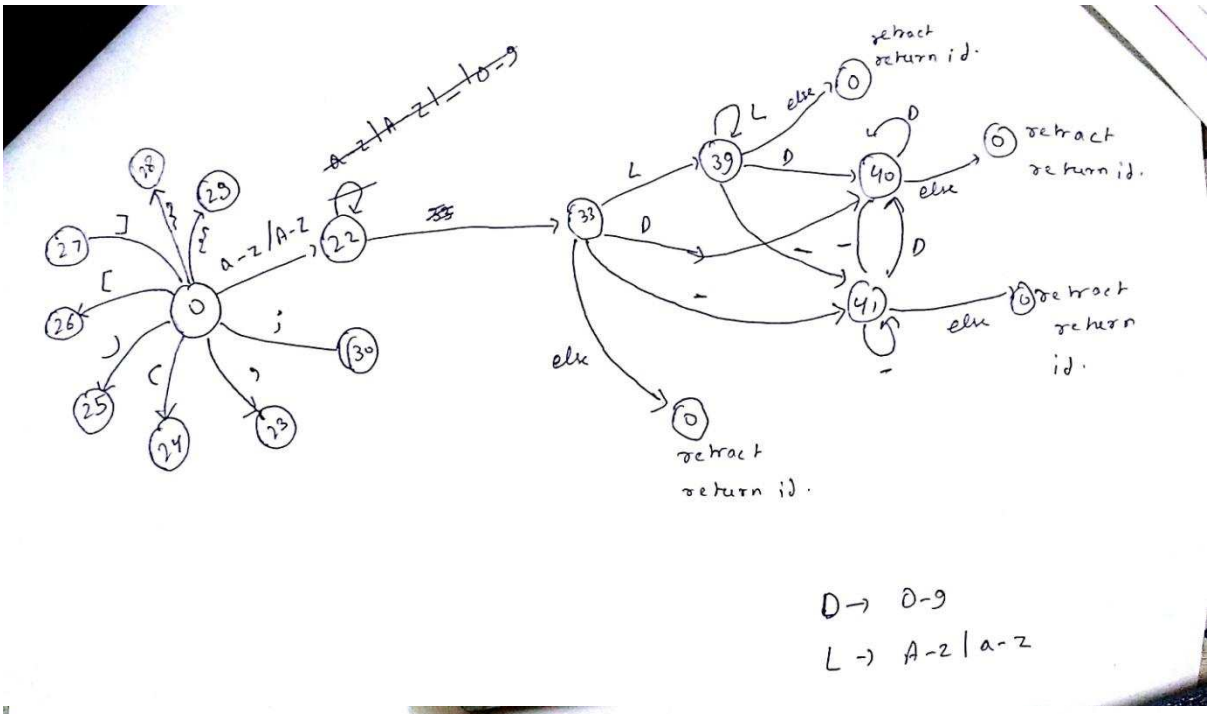
D1 -> M D1 | null

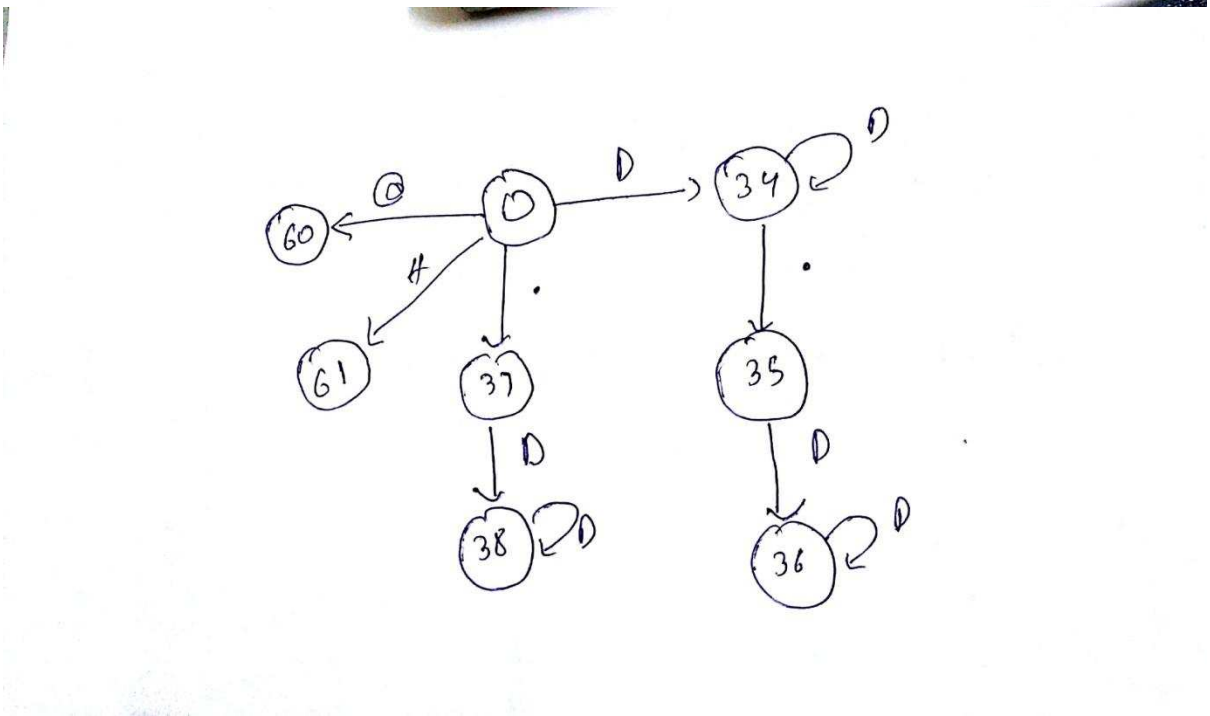
M -> *G | / G | %G

G-> ! G | id | TYPE

Pictures of the DFA:-







Sample Programs and their output:-

We are attaching the sample test input files in the “Testfiles” folder and their corresponding output generated by parser in “Snapshot” folder.

Parser Used:- Recursive Descent parser

Printing the output:- First we have to run the “lexer.c” it takes a sample program as a input file and generate all the tokens, then run the “RecursiveDescentParser.c” it reads the output given by “lexer.c” one token at a time. When we run “parser.c” it prints the corresponding rule which is used for the reduction on terminal(basically printing the stack).