1.

```
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{
   FILE *input;
   FILE *output;
   int l = 1;
   int t = 0;
   int j = 0;
   int i, flag = 0;
   char ch, str[20], y;
   input = fopen("input.txt", "r");
   output = fopen("output.txt", "w");
   char header[12][30] = {"util", "xml", "sql", "nio", "math", "awt", "net", "io", "lang", "security",
"java", "Scanner"};
   char keyword[44][30] = {"int", "class", "void", "import", "char", "abstract", "boolean",
"public", "static", "System", "private", "protected", "main", "String", "break", "byte", "case",
"catch", "class", "continue", "default", "do", "else", "enum", "exports", "extends", "final", "float",
"implements", "interface", "instanceof", "long", "new", "private", "public", "protected", "return",
"short", "super", "this", "throw", "try", "var", "void"};
   char conditionals[4][30] = {"if", "else", "switch", "case"};
   char looping[3][30] = {"do", "while", "for"};
   fprintf(output, "Line no. \t Token no. \t \tToken \t\t\t Lexeme\n\n");
   while (!feof(input))
   {
      i = 0;
      flag = 0;
      ch = fgetc(input);
      if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%')
      {
         fprintf(output, "%7d\t\t %7d\t\t Operator\t\t %7c\n", l, t, ch);
         t++;
      }
      if (ch == '?' || ch == ':')
      {
         fprintf(output, "%7d\t\t %7d\t\t Ternary Operator %7c\n", l, t, ch);
         t++;
      }
      else if (ch == '.')
      {
         fprintf(output, "%7d\t\t %7d\t\t Access Operator %7c\n", l, t, ch);
         t++;
      }
      else if (ch == '#')
      {
```

```
    fprintf(output, "%7d\t\t %7d\t\t Preprocessor\t %7c\n", l, t, ch);
    t++;
}

else if (ch == ';')
{
    fprintf(output, "%7d\t\t %7d\t\t End of Line\t %7c\n", l, t, ch);
    t++;
}
else if (ch == '{')
{
    fprintf(output, "%7d\t\t %7d\t\t Open braces\t %7c\n", l, t, ch);
    t++;
}
else if (ch == '}')
{
    fprintf(output, "%7d\t\t %7d\t\t Closing braces\t %7c\n", l, t, ch);
    t++;
}
else if (ch == '(' || ch == '[')
{
    fprintf(output, "%7d\t\t %7d\t\t Open brackets\t %7c\n", l, t, ch);
    t++;
}
else if (ch == ')' || ch == ']')
{
    fprintf(output, "%7d\t\t %7d\t\t Closing brackets %6c\n", l, t, ch);
    t++;
}
else if (ch == '"' || ch == '\'')
{
    fprintf(output, "%7d\t\t %7d\t\t Quotation marks %6c\n", l, t, ch);
    t++;
}
else if (isdigit(ch))
{
    fprintf(output, "%7d\t\t %7d\t\t Digit\t\t\t %7c\n", l, t, ch);
    t++;
}

else if (isalpha(ch) || ch == '$' || ch == '_')
{
    str[i] = ch;
    i++;
    ch = fgetc(input);
    while (isalnum(ch) || ch == '_' && ch != ' ')
    {
        str[i] = ch;
```

```
    i++;
    ch = fgetc(input);
}
str[i] = '\0';

if (flag == 0)
{
    for (j = 0; j <= 11; j++)
    {
        if (strcmp(str, header[j]) == 0)
        {
            flag = 2;
            break;
        }
    }
}
if (flag == 0)
{
    for (j = 0; j <= 3; j++)
    {
        if (strcmp(str, conditionals[j]) == 0)
        {
            flag = 3;
            break;
        }
    }
}
if (flag == 0)
{
    for (j = 0; j <= 2; j++)
    {
        if (strcmp(str, looping[j]) == 0)
        {
            flag = 4;
            break;
        }
    }
}
if (flag == 0)
{
    for (j = 0; j <= 43; j++)
    {
        if (strcmp(str, keyword[j]) == 0)
        {
            flag = 1;
            break;
        }
    }
```

```
            }
            if (flag == 1)
            {
                fprintf(output, "%7d\t\t %7d\t\t Keyword\t\t %7s\n", l, t, str);
                t++;
            }
            else if (flag == 2)
            {
                fprintf(output, "%7d\t\t %7d\t\t Header\t\t\t %7s\n", l, t, str);
                t++;
            }
            else if (flag == 3)
            {
                fprintf(output, "%7d\t\t %7d\t\t Conditional Statements %2s\n", l, t, str);
                t++;
            }
            else if (flag == 4)
            {
                fprintf(output, "%7d\t\t %7d\t\t Looping Statements %5s\n", l, t, str);
                t++;
            }
            else if (flag == 0)
            {
                if (strcmp(str, "instanceof") == 0)
                    fprintf(output, "%7d\t\t %7d\t\t Relational Operator\t\t %7s\n", l, t, str);
                else
                    fprintf(output, "%7d\t\t %7d\t\t Identifier\t\t %7s\n", l, t, str);
                t++;
            }
            flag = 0;
        }
        else if (ch == '\n')
        {
            l++;
        }
    }
    fclose(input);
    fclose(output);
    return 0;
}
```

3.

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
```

```
// Maximum number of states and symbols
#define MAX_STATES 10
#define MAX_SYMBOLS 10
// Function to build the DFA Transition Table
void buildTransitionTable(int states, int symbols, int
transitionTable[MAX_STATES][MAX_SYMBOLS])
{
   printf("Enter the transition states for each transition:\n");
   for (int i = 0; i < states; i++)
   {
      for (int j = 0; j < symbols; j++)
      {
         printf("Transition from state %d with input symbol %c (or 'null' if no transition): ", i, 'a'
+ j);
         char input[100];
         scanf("%s", input);
         if (strcmp(input, "null") == 0)
         {
            transitionTable[i][j] = -1;
         }
         else
         {
            sscanf(input, "%d", &transitionTable[i][j]);
         }
      }
   }
}
// Function to check if the input string is accepted by DFA
bool isAccepted(char *input, int states, int symbols, int transitionTable[][MAX_SYMBOLS],
bool acceptingStates[])
{
   int currentState = 0; // Start with initial state
   int len = strlen(input);
   printf("State Path: (%d, -) ", currentState); // Print the initial state
   for (int i = 0; i < len; i++)
   {
      int symbol = input[i] - 'a'; // Assuming input symbols are 'a' and 'b'
      if (symbol < 0 || symbol >= symbols)
      {
         printf("\nInvalid symbol found in the input!\n");
         return false;
      }
      int nextState = transitionTable[currentState][symbol];
      printf("-> (%d, %c) ", nextState, input[i]); // Print the state and input transition
      if (nextState < -1 || nextState >= states)
      {
         printf("\nInvalid state transition!\n");
         return false;
```

```
        }
        currentState = nextState;
    }
    printf("\n");
    // Print the transition table

    printf("\nTransition Table:\n");
    printf("State\t|");
    for (int j = 0; j < symbols; j++)
    {
        printf(" %c |", 'a' + j);
    }
    printf("\n");
    for (int i = 0; i < states; i++)
    {
        printf(" %d\t\t|", i);
        for (int j = 0; j < symbols; j++)
        {
            if (transitionTable[i][j] == -1)
            {
                printf(" null |");
            }
            else
            {
                printf(" %d |", transitionTable[i][j]);
            }
        }
        printf("\n");
    }
    return acceptingStates[currentState];
}
int main()
{
    int states, symbols;
    printf("Enter the number of states: ");
    scanf("%d", &states);
    printf("Enter the number of input symbols: ");
    scanf("%d", &symbols);
    // Validate the number of states and symbols
    if (states <= 0 || states > MAX_STATES || symbols <= 0 || symbols > MAX_SYMBOLS)
    {
        printf("Invalid number of states or symbols!\n");
        return 1;
    }
    int transitionTable[MAX_STATES][MAX_SYMBOLS];
    bool acceptingStates[MAX_STATES];
    buildTransitionTable(states, symbols, transitionTable);
    printf("\nEnter the input string (containing only 'a' to '%c'): ", 'a' + symbols - 1);
```

```
    char input[100];
    scanf("%s", input);
    bool accepted = isAccepted(input, states, symbols, transitionTable, acceptingStates);
    if (accepted)
        printf("String is accepted by the DFA!\n");
    else
        printf("String is not accepted by the DFA!\n");
    return 0;
}
```

5.

```
 #include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define MAX_RULES 10

// Grammar rule structure
typedef struct {
char nonTerminal;
char production[50];
} Rule;

// Function to match a specific terminal symbol
bool match(char symbol, char* input, int* pos) {
if (input[*pos] == symbol) {
(*pos)++;
return true;
}
return false;
}

// Recursive descent parsing function
bool parseInput(char* input, int* pos, Rule* grammar, int numRules, char
nonTerminal) {
for (int i = 0; i < numRules; i++) {
if (grammar[i].nonTerminal == nonTerminal) {
int oldPos = *pos;
bool valid = true;
for (int j = 0; grammar[i].production[j] != '\0'; j++) {
char symbol = grammar[i].production[j];
if (symbol >= 'A' && symbol <= 'Z') {
if (!parseInput(input, pos, grammar, numRules, symbol)) {
valid = false;
```

```
break;
}
} else {
if (!match(symbol, input, pos)) {
valid = false;
break;
}
}
}
if (valid) {
return true;
}
*pos = oldPos;
}
}
return false;
}

int main() {
char input[100];
Rule grammar[MAX_RULES];
int numRules;

printf("Enter the input string: ");
scanf("%s", input);

printf("Enter the number of grammar rules: ");
scanf("%d", &numRules);

printf("Enter the grammar rules in the format 'NonTerminal -> Production':\n");
for (int i = 0; i < numRules; i++) {
scanf(" %c -> %[^\n]", &grammar[i].nonTerminal, grammar[i].production);
}

int pos = 0;
char startSymbol = grammar[0].nonTerminal;

if (parseInput(input, &pos, grammar, numRules, startSymbol) && input[pos] == '\0')
{
printf("Input can be parsed.\n");
} else {
printf("Input cannot be parsed.\n");
}

return 0;
```

```
}
```

7

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SYMBOLS 100
#define MAX_RULES 100

typedef struct {
char lhs;
char rhs[MAX_SYMBOLS];
} Rule;

typedef struct {
Rule rules[MAX_RULES];
int num_rules;
} Grammar;

void read_grammar(Grammar* grammar) {
printf("Enter the number of rules: ");
scanf("%d", &grammar->num_rules);
printf("Enter the rules in the format A -> B1B2...Bn:\n");
for (int i = 0; i < grammar->num_rules; i++) {
scanf(" %c -> %s", &grammar->rules[i].lhs, grammar->rules[i].rhs);
}
}

void print_grammar(Grammar grammar) {
printf("Grammar:\n");
for (int i = 0; i < grammar.num_rules; i++) {
printf("%c -> %s\n", grammar.rules[i].lhs, grammar.rules[i].rhs);
}
}

int is_non_terminal(char symbol) {
return symbol >= 'A' && symbol <= 'Z';
}

void compute_first(Grammar grammar, char* variable, char* first) {
```

```
first[0] = '\0';
if (!is_non_terminal(variable[0])) {
strncat(first, variable, 1);
return;
}
for (int i = 0; i < grammar.num_rules; i++) {
if (grammar.rules[i].lhs == variable[0]) {
char* rhs = grammar.rules[i].rhs;
if (rhs[0] == 'e') {
strncpy(first, "e", 1);
}
else {
char sub_first[MAX_SYMBOLS];
compute_first(grammar, rhs, sub_first);
for (int j = 0; j < strlen(sub_first); j++) {
if (sub_first[j] != 'e' && strchr(first, sub_first[j]) == NULL) {
strncat(first, &sub_first[j], 1);
}
}
}
}
}
}

void compute_follow(Grammar grammar, char variable, char* follow) {
char sub_follow[MAX_SYMBOLS];
follow[0] = '\0';
if (!is_non_terminal(variable)) {
return;
}
if (variable == grammar.rules[0].lhs) {
strncat(follow, "$", 2);
}
for (int i = 0; i < grammar.num_rules; i++) {
char* rhs = grammar.rules[i].rhs;
for (int j = 0; j < strlen(rhs); j++) {
if (rhs[j] == variable) {
if (j == strlen(rhs)-1) {
compute_follow(grammar, grammar.rules[i].lhs, sub_follow);
for (int k = 0; k < strlen(sub_follow); k++) {
if (strchr(follow, sub_follow[k]) == NULL) {
strncat(follow, &sub_follow[k], 1);
}
}
}
}
```

```
else {
if (!is_non_terminal(rhs[j+1])) {
strncat(follow, &rhs[j+1], 1);
}
else {
char sub_first[MAX_SYMBOLS];
compute_first(grammar, &rhs[j+1], sub_first);
for (int k = 0; k < strlen(sub_first); k++) {
if (sub_first[k] != 'e' && strchr(follow, sub_first[k]) == NULL) {
strncat(follow, &sub_first[k], 1);
}
}
if (strchr(sub_first, 'e') != NULL) {
compute_follow(grammar, variable, sub_follow);
for (int k = 0; k < strlen(sub_follow); k++) {
if (strchr(follow, sub_follow[k]) == NULL) {
strncat(follow, &sub_follow[k], 1);
}
}
}
}
}
}
}
}
}

int main() {
Grammar grammar;
read_grammar(&grammar);
print_grammar(grammar);

char variable;
printf("Enter a variable to compute first/follow for: ");
scanf(" %c", &variable);

int choice;
printf("Enter 1 to compute FIRST or 2 to compute FOLLOW: ");
scanf("%d", &choice);

char result[MAX_SYMBOLS];
if (choice == 1) {
compute_first(grammar, &variable, result);
printf("First(%c) = {%s}\n", variable, result);
}
```

```
else if (choice == 2) {
compute_follow(grammar, variable, result);
printf("Follow(%c) = {%s}\n", variable, result);
}
else {
printf("Invalid choice\n");
}

return 0;
}
```

9

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct three
{
   char data[10], temp[7];
} s[30];
int main()
{
   char d1[7], d2[7] = "t";
   int i = 0, j = 1, len = 0;
   FILE *f1, *f2;
   f1 = fopen("sum.txt", "r");
   f2 = fopen("out.txt", "w");
   while (fscanf(f1, "%s", s[len].data) != EOF)
      len++;
   (void)sprintf(d1, "%d", j);
   strcat(d2, d1);
   strcpy(s[j].temp, d2);
   strcpy(d1, "");
   strcpy(d2, "t");
   if (!strcmp(s[3].data, "+"))
   {
      fprintf(f2, "%s=%s+%s", s[j].temp, s[i + 2].data, s[i + 4].data);
      j++;
   }
   else if (!strcmp(s[3].data, "-"))
```

```
    {
        fprintf(f2, "%s=%s-%s", s[j].temp, s[i + 2].data, s[i + 4].data);
        j++;
    }
    for (i = 4; i < len - 2; i += 2)
    {
        (void)sprintf(d1, "%d", j);
        strcat(d2, d1);
        strcpy(s[j].temp, d2);
        if (!strcmp(s[i + 1].data, "+"))
            fprintf(f2, "\n%s=%s+%s", s[j].temp, s[j - 1].temp, s[i + 2].data);
        else if (!strcmp(s[i + 1].data, "-"))
            fprintf(f2, "\n%s=%s-%s", s[j].temp, s[j - 1].temp, s[i + 2].data);
        else if (!strcmp(s[i + 1].data, "*"))
            fprintf(f2, "\n%s=%s*%s", s[j].temp, s[j - 1].temp, s[i + 2].data);
        else if (!strcmp(s[i + 1].data, "/"))
            fprintf(f2, "\n%s=%s/%s", s[j].temp, s[j - 1].temp, s[i + 2].data);
        strcpy(d1, "");
        strcpy(d2, "t");
        j++;
    }
    fprintf(f2, "\n%s=%s", s[0].data, s[j - 1].temp);
    fclose(f1);
    fclose(f2);
    return 0;
}
```

Lex

Design simple calculator using lex tool to solve arithmetic expression

CODE:

```
%{
 #include <stdio.h>
```

```
#include <stdlib.h>
void digi();
int op = 0, i;
float a, b;
%}
%%
[0-9]+|([0-9]+"."[0-9]+) { digi(); }
"+" { op = 1; }
"-" { op = 2; }
"*" { op = 3; }
"/" { op = 4; }
"^" { op = 5; }
\n { printf("\n The Answer: %f\n\n", a); }
```

```
%%
 void digi()


{if (op == 0)
 a = atof(yytext); else


{b = atof(yytext); switch (op)


{case 1:
 a = a + b;
 break;
 case 2:
 a = a - b;
 break;
```

```
 case 3:
 a = a * b;
 break;
 case 4:
 a = a / b;
 break;
 case 5:
 for (i = a; b > 1; b--) a = a * i;
 break;


}o p = 0 ;


}int main(int argc, char *argv[])


{yylex(); return 0;


}int yywrap()


{return 1; }
```

Lex Program to check whether a number is Prime or Not

CODE:

```
%{ #include<stdio.h> #include<stdlib.h> int flag,c,j;
 %}


%%
 [0-9]+ {c=atoi(yytext);


if(c==2)
 {printf("\n Prime number"); }else if(c==0 || c==1)







{printf("\n Not a Prime number"); }else
 {for(j=2;j<c;j++)
 {if(c%j==0)


flag=1;


}if(flag==1)
 printf("\n Not a prime number"); else if(flag==0)
 printf("\n Prime number");
 }


} %%


int main()
```

```
{yylex(); return 0; }
```