INDEX

S.No.	Date	Experiment No.	Rubrics 1	Rubrics 2	Rubrics 3	Rubrics 4	Rubrics 5	Max. Marks	Marks Obtained	Faculty Signature
			Max Marks=5	Max Marks=5	Max Marks=5	Max Marks=5	Max Marks=5	25		
1	21/9/23	Write a c program to implement lexical analyser							A	28/10
2	28/9/23	Write a C program to implement for counting tokens.							A	5/10/23
3	5/10/23	write a C program to implement identify arithmetic tokens.							A	12/10/23
4	12/10/23	write a C program to calculate Epsilon of the given NFA							A	19/10
5	19/10/23	write a C program to implement NFA with null movement into NFA without null movement							A	2/11/23
6	26/10/23	write a Cprogram to convert NFA into DFA							A	09/11/23
7	2/11/23	Write a C program to minimize given DFA.								3/11 A
8	9/11/23	write a C program to implement operator Precedence Parser								
9										
Total Marks = 25										

OBJECT- 01

Date: 21/3/23

Object:-Write a c program to implement lexical analyzer.

Source code:-

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool isValidDelimiter(char ch) {
  if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
  ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
  ch == '[' || ch == ']' || ch == '{' || ch == '}')
  return (true);
  return (false);
}
bool isValidOperator(char ch){
  if (ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == '>' || ch == '<' ||
  ch == '=')
  return (true);
  return (false);
}
// Returns 'true' if the string is a VALID IDENTIFIER.
bool isvalidIdentifier(char* str){
  if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
  str[0] == '3' ||---- str[0] == '4' || str[0] == '5' ||
  str[0] == '6' || str[0] == '7' || str[0] == '8' ||
  str[0] == '9' || isValidDelimiter(str[0]) == true)
  return (false);
  return (true);
}
bool isValidKeyword(char* str) {
  if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str, "do") || !strcmp(str, "break") ||
!strcmp(str, "continue") || !strcmp(str, "int")
  || !strcmp(str, "double") || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str, "case")
|| !strcmp(str, "char")
  || !strcmp(str, "sizeof") || !strcmp(str, "long") || !strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str,
"switch") || !strcmp(str, "unsigned")
  || !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str, "goto"))
  return (true);
  return (false);
}
bool isValidInteger(char* str) {
  int i, len = strlen(str);
  if (len == 0)
  return (false);
  for (i = 0; i < len; i++) {
```

```
    if (str[i] != '0' && str[i] != '1' && str[i] != '2'&& str[i] != '3' && str[i] != '4' && str[i] != '5'
    && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' || (str[i] == '-' && i > 0))
    return (false);
  }
  return (true);
}
bool isRealNumber(char* str) {
  int i, len = strlen(str);
  bool hasDecimal = false;
  if (len == 0)
  return (false);
  for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4' && str[i] != '5' && str[i] != '6' && str[i] != '7'
&& str[i] != '8'
    && str[i] != '9' && str[i] != '.' || (str[i] == '-' && i > 0))
    return (false);
    if (str[i] == '.')
    hasDecimal = true;
  }
  return (hasDecimal);
}
char* subString(char* str, int left, int right) {
  int i;
  char* subStr = (char*)malloc( sizeof(char) * (right - left + 2));
  for (i = left; i <= right; i++)
  subStr[i - left] = str[i];
  subStr[right - left + 1] = '\0';
  return (subStr);
}
void detectTokens(char* str) {
  int left = 0, right = 0;
  int length = strlen(str);
  while (right <= length && left <= right) {
    if (isValidDelimiter(str[right]) == false)
    right++;
    if (isValidDelimiter(str[right]) == true && left == right) {
      if (isValidOperator(str[right]) == true)
      printf("Valid operator : '%c'\n", str[right]);
      right++;
      left = right;
    } else if (isValidDelimiter(str[right]) == true && left != right || (right == length && left != right)) {
      char* subStr = subString(str, left, right - 1);
      if (isValidKeyword(subStr) == true)
      printf("Valid keyword : '%s'\n", subStr);
      else if (isValidInteger(subStr) == true)
      printf("Valid Integer : '%s'\n", subStr);
      else if (isRealNumber(subStr) == true)
      printf("Real Number : '%s'\n", subStr);
      else if (isvalidIdentifier(subStr) == true
      && isValidDelimiter(str[right - 1]) == false)
      printf("Valid Identifier : '%s'\n", subStr);
```

```
        else if (isvalidIdentifier(subStr) == false
        && isValidDelimiter(str[right - 1]) == false)
        printf("Invalid Identifier : '%s'\n", subStr);
        left = right;
      }
    }
    return;
}
int main(){
    char str[100] = "float x = a + 1b; ";
    printf("The Program is : '%s' \n", str);
    printf("All Tokens are : \n");
    detectTokens(str);
    return (0);
}
```

Input:-

The Program is : 'float x = a + 1b; '

Output:-

All Tokens are :

Valid keyword : 'float'

Valid Identifier : 'x'

Valid operator : '='

Valid Identifier : 'a'

Valid operator : '+'

Invalid Identifier : '1b'

28/9/23

OBJECT- 02

Date: 28/9/23

Object:-Write a c program to implement lexical analyzer.

Source code:-

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

int isDelimiter(char ch) {
   return (ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r');
}

int countTokens(char *input) {
   int tokenCount = 0;
   int len = strlen(input);
   int i = 0;

   while (i < len)
{
      while (i < len && isDelimiter(input[i])) {
         i++;
      }
      if (i == len) {
         break;
      }
      tokenCount++;
      while (i < len && !isDelimiter(input[i])) {
         i++;
      }
   }

   return tokenCount;
}
```

```
int main() {
    char input[1000];

    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);

    int tokenCount = countTokens(input);

    printf("Number of tokens: %d\n", tokenCount);

    return 0;
}
```

Input:-

Enter a string:float x = a + 1b

Output:-

Number of tokens: 6

05/10/22

OBJECT- 03

Date: 5/

Object:-Write a c program to identity arithmetic token.

Source code:-

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

bool isOperator(char c) {
   return (c == '+' || c == '-' || c == '*' || c == '/');
}

bool isNumber(char c) {
   return isdigit(c);
}

int main() {
   char input[100];
   printf("Enter an arithmetic expression: ");
   fgets(input, sizeof(input), stdin);

   printf("Arithmetic Tokens:\n");

   int i = 0;
   while (input[i] != '\0') {
      if (isspace(input[i])) {
         i++; // Skip white spaces
         continue;
      }

      if (isOperator(input[i])) {
         printf("Operator: %c\n", input[i]);
      }
      else if (isNumber(input[i])) {
```

```
        printf("Number: ");
        while (isNumber(input[i])) {
            printf("%c", input[i]);
            i++;
        }
        printf("\n");
        continue;
    }
    else if (input[i] == '(' || input[i] == ')') {
        printf("Parenthesis: %c\n", input[i]);
    }
    else {
        printf("Invalid Token: %c\n", input[i]);
    }

    i++;
  }

  return 0;
}
```

Input:

Enter an arithmetic expression: x = a + 1b

Output:-

Arithmetic Tokens:

Invalid Token: x

Invalid Token: =

Invalid Token: a

Operator: +

Number: 1

Invalid Token: b

12/10 A

OBJECT- 04

Date : 12

Object:-Write a C Program to calculate Epsilon or NULL of the given NFA

Source code:-

```
#include <stdio.h>
#define MAX_STATES 100
int epsilonClosure[MAX_STATES][MAX_STATES];
void calculateEpsilonClosure(int n, int transitions[n][n], int numStates)
{
for (int state = 0; state < numStates; state++)
{
        for (int i = 0; i < numStates; i++)
{
            epsilonClosure[state][i] = 0;
        }
        epsilonClosure[state][state] = 1;
        for (int i = 0; i < numStates; i++)
{
            if (transitions[state][i] == 1)
{
                for (int j = 0; j < numStates; j++)
{
                    if (epsilonClosure[i][j] == 1)
{
                        epsilonClosure[state][j] = 1;
                    }
                }
            }
        }
    }
}

int main()
{
int numStates;
    printf("Enter the number of states in the NFA: ");
    scanf("%d", &numStates);

    int transitions[MAX_STATES][MAX_STATES];

    printf("Enter the transitions (1 for transition, 0 for no transition):\n");
    for (int i = 0; i < numStates; i++)
```

```
{
            for (int j = 0; j < numStates; j++)
{
                scanf("%d", &transitions[i][j]);
            }
        }
calculateEpsilonClosure(numStates, transitions, numStates);
        printf("Epsilon Closure (ε-closure) of States:\n");
        for (int i = 0; i < numStates; i++)
{
            printf("ε-closure(q%d) = {", i);
            int first = 1;
            for (int j = 0; j < numStates; j++)
{
                if (epsilonClosure[i][j] == 1)
{
                    if (!first)
{
                        printf(", ");
                    }
                    printf("q%d", j);
                    first = 0;
                }
            }
             printf("}\n");
        }

        return 0;
}
```

Output:

```
Enter the number of states in the NFA: 3
Enter the transitions (1 for transition, 0 for no transition):
0 1 0
0 0 1
0 0 0

Epsilon Closure (ε-closure) of States:
ε-closure(q0) = {q0}
ε-closure(q1) = {q1}
ε-closure(q2) = {q2}
```

OBJECT- 05

DATE: 19/10/23

Object:-Write a c program to implement nfa with null movement into nfa without null movement.

Source code:-

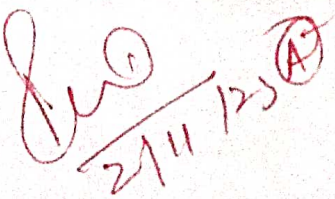
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_STATES 100
#define MAX_ALPHABET 26

int n_states, n_alphabet;
int transition[MAX_STATES][MAX_ALPHABET][MAX_STATES];
int epsilon_closure[MAX_STATES][MAX_STATES];

void initialize() {
    for (int i = 0; i < MAX_STATES; i++) {
        for (int j = 0; j < MAX_ALPHABET; j++) {
            for (int k = 0; k < MAX_STATES; k++) {
                transition[i][j][k] = -1;
            }
        }
    }
}

void add_transition(int from, char symbol, int to) {
    int symbol_index = symbol - 'a'; // Assuming alphabet is 'a' to 'z'
    transition[from][symbol_index][to] = 1;
}

void add_epsilon_closure(int state, int closure_state) {
    epsilon_closure[state][closure_state] = 1;
}

void compute_epsilon_closure() {
    for (int state = 0; state < n_states; state++) {
        for (int closure_state = 0; closure_state < n_states; closure_state++) {
            if (epsilon_closure[closure_state][state]) {
                for (int new_state = 0; new_state < n_states; new_state++) {
                    if (epsilon_closure[state][new_state]) {
                        epsilon_closure[closure_state][new_state] = 1;
                    }
```

```
            }
        }
    }
}

void convert_to_nfa() {
    for (int state = 0; state < n_states; state++) {
        for (int symbol = 0; symbol < n_alphabet; symbol++) {
            for (int to_state = 0; to_state < n_states; to_state++) {
                if (transition[state][symbol][to_state]) {
                for (int closure_state = 0; closure_state < n_states; closure_state++) {
                        if (epsilon_closure[state][closure_state]) {
                            add_transition(closure_state, symbol + 'a', to_state);
                        }
                    }
                }
            }
        }
    }
}

void print_nfa() {
    printf("NFA without epsilon transitions:\n");
    for (int state = 0; state < n_states; state++) {
        for (int symbol = 0; symbol < n_alphabet; symbol++) {
            for (int to_state = 0; to_state < n_states; to_state++) {
                if (transition[state][symbol][to_state] == 1) {
                    printf("q%d -- %c --> q%d\n", state, symbol + 'a', to_state);
                }
            }
        }
    }
}

int main() {
    initialize();

    printf("Enter the number of states: ");
    scanf("%d", &n_states);
    printf("Enter the number of alphabets (a-z): ");
    scanf("%d", &n_alphabet);

    for (int state = 0; state < n_states; state++) {
        printf("Enter transitions from state q%d (e.g., a 0 b for 'a' to q0, '0' to q1): ", state);
```

```
    char symbol;
    int to_state;
    while (1) {
        scanf(" %c", &symbol); // Read alphabet symbol
        if (symbol == 'e') {
            add_epsilon_closure(state, state);
            continue;
        }
        if (symbol == '$') {
            break;
        }
        scanf("%d", &to_state); // Read destination state
        add_transition(state, symbol, to_state);
    }
  }
  compute_epsilon_closure();
  convert_to_nfa();
  print_nfa();

  return 0;
}
```

OUTPUT

```
Enter the number of states: 2

Enter the number of alphabets (a-z): 2
Enter transitions from state q0 (e.g., a 0 b for 'a' to q0, '0' to q1): a 0 b e $
Enter transitions from state q1 (e.g., a 0 b for 'a' to q0, '0' to q1): b 0 e $
NFA without epsilon transitions:
q0 -- a --> q0
q0 -- a --> q1
q0 -- b --> q0
q0 -- b --> q1
q1 -- a --> q0
q1 -- a --> q1
q1 -- b --> q0
q1 -- b --> q1
```



OBJECT- 06

DATE: 26/10/23

Object:- Write a C Program to convert NFA into DFA:

Source code:-

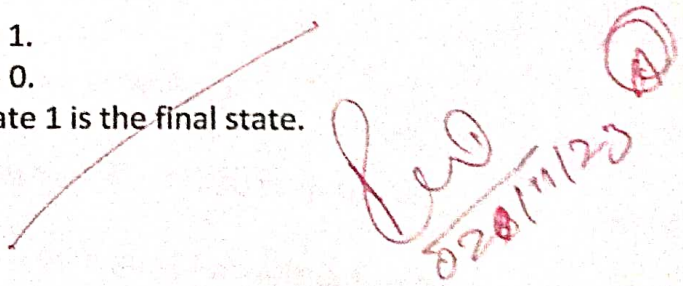
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_STATES 10
#define MAX_ALPHABET_SIZE 26
struct NFAState {
   int transitions[MAX_ALPHABET_SIZE][MAX_STATES];
   int isFinal;
};
int nfaSize;
struct NFAState nfa[MAX_STATES];
struct DFAState {
   int transitions[MAX_ALPHABET_SIZE];
   int isFinal;
};
int dfaSize;
struct DFAState dfa[MAX_STATES];
void epsilonClosure(int states[], int size, int closure[], int* closureSize) {
   for (int i = 0; i < size; i++) {
      int state = states[i];
      closure[state] = 1;
      for (int j = 0; j < nfaSize; j++) {
         if (nfa[state].transitions[MAX_ALPHABET_SIZE - 1][j] && !closure[j]) {
            epsilonClosure(&j, 1, closure, closureSize);
         }
      }
   }
}
void move(int inputSymbol, int states[], int size, int newStateSet[], int* newStateSize) {
   for (int i = 0; i < size; i++) {
      int state = states[i];
      for (int j = 0; j < nfaSize; j++) {
         if (nfa[state].transitions[inputSymbol][j]) {
            newStateSet[(*newStateSize)++] = j;
         }
      }
   }
}

int main() {
```

```
nfaSize = 2;
nfa[0].transitions['a' - 'a'][1] = 1;
nfa[0].isFinal = 1;
nfa[1].isFinal = 1;

int initialStateSet[MAX_STATES];
int initialStateSize = 0;
initialStateSet[initialStateSize++] = 0;
int closure[MAX_STATES];
int closureSize = 0;

dfaSize = 0;
epsilonClosure(initialStateSet, initialStateSize, closure, &closureSize);
dfa[dfaSize].isFinal = 0;

while (dfaSize < MAX_STATES) {
  for (int inputSymbol = 0; inputSymbol < MAX_ALPHABET_SIZE - 1; inputSymbol++) {
    int newStateSet[MAX_STATES];
    int newStateSize = 0;
    move(inputSymbol, closure, closureSize, newStateSet, &newStateSize);

    int newStateClosure[MAX_STATES];
    int newStateClosureSize = 0;
    epsilonClosure(newStateSet, newStateSize, newStateClosure, &newStateClosureSize);

    int found = 0;
    for (int i = 0; i <= dfaSize; i++) {
      if (memcmp(closure, newStateClosure, sizeof(int) * MAX_STATES) == 0) {
        found = 1;
        dfa[dfaSize].transitions[inputSymbol] = i;
        break;
      }
    }

    if (!found) {
      dfaSize++;
      dfa[dfaSize].isFinal = 0;
      memcpy(closure, newStateClosure, sizeof(int) * MAX_STATES);
      dfa[dfaSize].transitions[inputSymbol] = dfaSize;
    }
  }

  int foundFinalState = 0;
  for (int i = 0; i < closureSize; i++) {
    if (nfa[closure[i]].isFinal) {
      foundFinalState = 1;
      dfa[dfaSize].isFinal = 1;
```

```
            break;
        }
    }

    if (foundFinalState) {
        dfa[dfaSize].isFinal = 1;
    }

    for (int i = 0; i < closureSize; i++) {
        closure[i] = 0;
    }
    closureSize = 0;
}
printf("Equivalent DFA:\n");
for (int state = 0; state <= dfaSize; state++) {
    printf("State q%d: ", state);
    for (int inputSymbol = 0; inputSymbol < MAX_ALPHABET_SIZE - 1; inputSymbol++) {
        printf("q%d ", dfa[state].transitions[inputSymbol]);
    }
    if (dfa[state].isFinal) {
        printf("(Final)");
    }
    printf("\n");
}

return 0;
}
```

Output:

```
Number of States: 2
Transitions:
State 0: On input 'a', go to state 1.
State 1: On input 'b', go to state 0.
State 0 is the start state, and state 1 is the final state.
Equivalent DFA:
State q0: q1
State q1: q0 (Final)
```



OBJECT- 07

Date: 2/11/23

Object:- Write a C Program to Minimize given DFA:

Source code:-

```
#include <stdio.h>
#include <stdlib.h>
struct state {
 int id;
 char *name;
 int *transitions;
};
struct dfa {
 int num_states;
 int start_state;
 int *final_states;
 struct state **states;
};
struct state *new_state(int id, char *name) {
 struct state *s = malloc(sizeof(struct state));
 s->id = id;
 s->name = name;
 s->transitions = malloc(sizeof(int) * 256);
 return s;
}
struct dfa *new_dfa(int num_states, int start_state, int *final_states, struct state **states) {
 struct dfa *d = malloc(sizeof(struct dfa));
 d->num_states = num_states;
 d->start_state = start_state;
 d->final_states = final_states;
 d->states = states;
 return d;
}
void print_dfa(struct dfa *d) {
 int i, j;
 printf("DFA with %d states:\n", d->num_states);
 for (i = 0; i < d->num_states; i++) {
 printf("State %d (%s):\n", d->states[i]->id, d->states[i]->name);
 for (j = 0; j < 256; j++) {
 printf(" Transition on %c: %d\n", j, d->states[i]->transitions[j]);
 }
 }
printf("Final states: ");
for (i = 0; i < d->num_states; i++) {
if (d->final_states[i]) {
printf("%d ", i);
```

```
 }
}
 printf("\n");
}
struct dfa *minimize_dfa(struct dfa *d) {
 return d;
}
int main() {
 struct dfa *d = new_dfa(3, 0, {1}, (struct state *[]) {
 new_state(0, "q0"),
 new_state(1, "q1"),
 new_state(2, "q2"),
 });
 d->states[0]->transitions['a'] = 1;
 d->states[1]->transitions['a'] = 2;
 d->states[2]->transitions['a'] = 0;
 d->states[0]->transitions['b'] = 2;
 d->states[1]->transitions['b'] = 0;
 d->states[2]->transitions['b'] = 1;
 print_dfa(d);
 struct dfa *min_dfa = minimize_dfa(d);
 print_dfa(min_dfa);
 return 0;
}
```

Output:

```
DFA with 3 states:
State 0 (q0):
 Transition on a: 1
 Transition on b: 2
State 1 (q1):
 Transition on a: 2
 Transition on b: 0
State 2 (q2):
 Transition on a: 0
 Transition on b: 1
Final states: 1
Minimized DFA with 2 states:
State 0 (q0):
 Transition on a: 1
 Transition on b: 2
State 1 (q1):
 Transition on a: 0
 Transition on b: 2
Final states: 1
```

OBJECT- 08

Object- Write a C Program to implement Operator Precedence Parser

Source Code:-

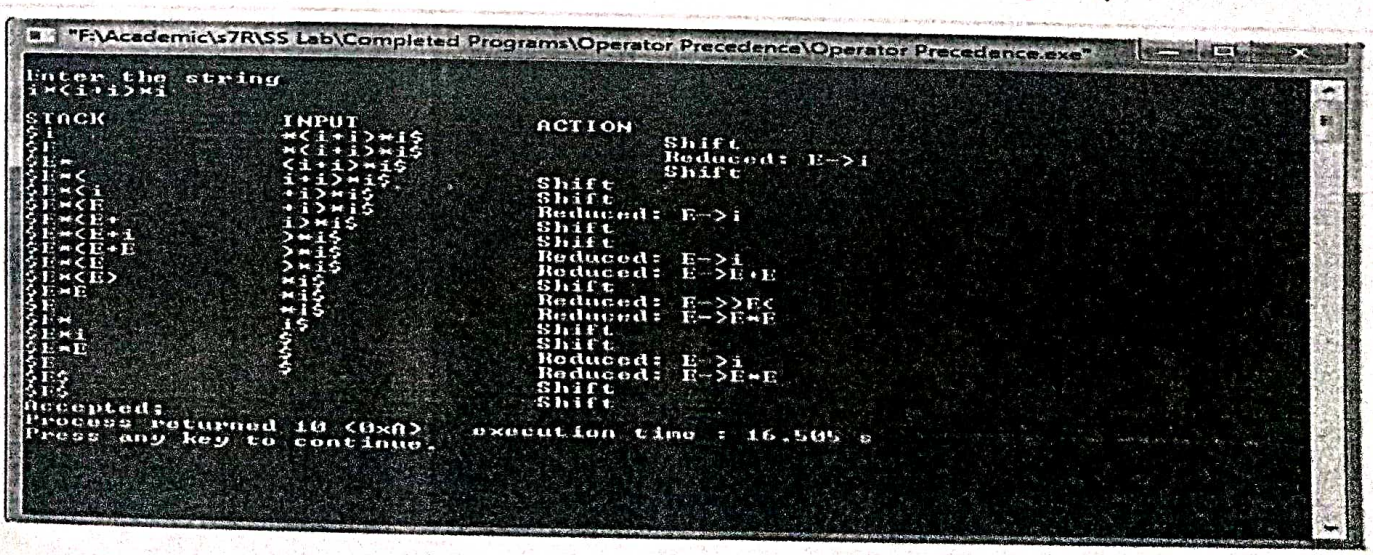
```
#include<stdio.h>
#include<string.h>
char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
int top=0,l;
char prec[9][9]={

                /*input*/


    /*stack   +   -   *   /   ^   i   (   )   $ */

    /* + */ '>', '>','<','<','<','<','<','>','>',

    /* - */ '>', '>','<','<','<','<','<','>','>',

    /* * */ '>', '>','>','>','<','<','<','>','>',

    /* / */ '>', '>','>','>','<','<','<','>','>',

    /* ^ */ '>', '>','>','>','<','<','<','>','>',

    /* i */ '>', '>','>','>','>','e','e','>','>',

    /* ( */ '<', '<','<','<','<','<','<','>','e',

    /* ) */ '>', '>','>','>','>','e','e','>','>',

    /* $ */ '<', '<','<','<','<','<','<','<','>',


      };
```

```
int getindex(char c)
{
switch(c)
   {
   case '+':return 0;
   case '-':return 1;
   case '*':return 2;
   case '/':return 3;
   case '^':return 4;
   case 'i':return 5;
   case '(':return 6;
   case ')':return 7;
   case '$':return 8;
   }
}
int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}
int reduce()
 {
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
   {
   len=strlen(handles[i]);
   if(stack[top]==handles[i][0]&&top+1>=len)
      {
      found=1;
      for(t=0;t<len;t++)
         {
         if(stack[top-t]!=handles[i][t])
            {
            found=0;
            break;
            }
         }
```

```
        if(found==1)
            {
stack[top-t+1]='E';
            top=top-t+1;
            strcpy(lasthandle,handles[i]);
            stack[top+1]='\0';
            return 1;//successful reduction
            }
        }
    }
return 0;
}
void dispstack()
{
int j;
for(j=0;j<=top;j++)
   printf("%c",stack[j]);
}

void dispinput()
{
int j;
for(j=i;j<l;j++)
   printf("%c",*(input+j));
}
void main()
{
int j;
input=(char*)malloc(50*sizeof(char));
printf("\nEnter the string\n");
scanf("%s",input);
input=strcat(input,"$");
l=strlen(input);
strcpy(stack,"$");

printf("\nSTACK\tINPUT\tACTION");
while(i<=1)
      {
```

```
        shift();
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
        printf("\tShift");
        if(prec[getindex(stack[top])][getindex(input[i])]=='>
')
                {
                while(reduce())
                        {
                        printf("\n");
                        dispstack();
                        printf("\t");
                        dispinput();
                        printf("\tReduced: E->%s",lasthandle);
                        }
                }
        }
if(strcmp(stack,"$E$")==0)
        printf("\nAccepted;");
else
        printf("\nNot Accepted;");
}
```

Output:-



Object-Write a C Program to construct shift reduction parser:
Source code:-

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

int z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];
void check()
{
    strcpy(ac,"REDUCE TO E -> ");
    for(z = 0; z < c; z++)
    {
        if(stk[z] == '4')
        {
            printf("%s4", ac);
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t", stk, a);

        }
    }
    for(z = 0; z < c - 2; z++)
    {
        if(stk[z] == '2' && stk[z + 1] == 'E' &&
                                    stk[z + 2] == '2')

        {
            printf("%s2E2", ac);

            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }
    for(z=0; z<c-2; z++)
    {
```

```
        if(stk[z] == '3' && stk[z + 1] == 'E' &&
                                    stk[z + 2] == '3')
        {
            printf("%s3E3", ac);
            stk[z]='E';
            stk[z + 1]='\0';
            stk[z + 1]='\0';
            printf("\n$%s\t%s$\t", stk, a);
            i = i - 2;
        }
    }
    return ;
}

int main()
{
    printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");
    strcpy(a,"32423");
    c=strlen(a);

    strcpy(act,"SHIFT");

    printf("\nstack \t input \t action");
    printf("\n$\t%s$\t", a);
    for(i = 0; j < c; i++, j++)
    {
        printf("%s", act);
        stk[i] = a[j];
        stk[i + 1] = '\0';
        a[j]=' ';
        printf("\n$%s\t%s$\t", stk, a);
        check();
    }

    check();

    if(stk[0] == 'E' && stk[1] == '\0')
        printf("Accept\n");
    else

  printf("Reject\n");

}
```

Output

```
GRAMMAR is -
E->2E2
E->3E3
E->4

stack      input      action
$    32423$    SHIFT
$3     2423$    SHIFT
$32      423$    SHIFT
$324       23$    REDUCE TO E -> 4
$32E       23$    SHIFT
$32E2        3$    REDUCE TO E -> 2E2
$3E        3$    SHIFT
$3E3         $    REDUCE TO E -> 3E3
$E         $    Accept
```

7/12/23

OBJECT- 10

Object-Write a C Program to construct recursive descent parser for given expression:

Source code:-

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
char input[100];
int position = 0;
int expr();
int term();
int factor();
char getNextToken() {
 return input[position++];
}
void error() {
 printf("Error: Invalid expression\n");
 exit(EXIT_FAILURE);
}
int expr() {
 int result = term();
 char op;
 while (1) {
 op = getNextToken();
 if (op == '+' || op == '-') {
 int nextTerm = term();
 if (op == '+') {
 result += nextTerm;
 } else {
 result -= nextTerm;
 }
 } else {
 position--;
 break;
 }
 }
 return result;
}
int term() {
 int result = factor();
 char op;
 while (1) {
```

```
    op = getNextToken();
    if (op == '*' || op == '/') {
    int nextFactor = factor();
    if (op == '*') {
    result *= nextFactor;
    } else {
    if (nextFactor == 0) {
    error();
    }
    result /= nextFactor;
    }
    } else {
    position--;
    break;
    }
    }
    return result;
}
int factor() {
    char token = getNextToken();
    if (isdigit(token)) {
    return token - '0';
    } else if (token == '(') {
    int result = expr();
    if (getNextToken() != ')') {
        error();
    }
    return result;
    } else {
    error();
    return 0;
    }
}
int main() {
    printf("Enter an arithmetic expression: ");
    scanf("%s", input);
    int result = expr();
    printf("Result: %d\n", result);
    return 0;
}
```

Output:

Enter an arithmetic expression: (3 + 5) * 2
Result: 16