1.Construction of NFA

```
#include <stdio.h>
#include <string.h>
int main() {
   int nfa[3][2][3] = {
      {{1,2,-1},{-1}},  // state 0
      {{-1},{2,-1}},    // state 1
      {{-1},{-1}}       };
   int final = 2, curr[10] = {0}, next[10], i, j, k, c = 1, n;
   char str[100];
   printf("Enter the string :\n");
   scanf("%s", str);
   printf("%c ", str[0]);
   for (i = 0; str[i]; i++) {
      int sym = str[i] - 'a', nc = 0;
      for (j = 0; j < c; j++) {
         int st = curr[j];
         for (k = 0; nfa[st][sym][k] != -1; k++) {
            int exists = 0;
            for (n = 0; n < nc; n++)
               if (next[n] == nfa[st][sym][k]) exists = 1;
            if (!exists)
               next[nc++] = nfa[st][sym][k]; }}
      if (nc == 0) {
         printf("-1\nSTRING NOT ACCEPTED\n");
         return 0;}
      for (j = 0; j < nc; j++) printf("%d ", next[j]);
      memcpy(curr, next, sizeof(next));
      c = nc;}
   for (i = 0; i < c; i++)
      if (curr[i] == final) {
         printf("\nSTRING ACCEPTED\n");
         return 0;}
   printf("\nSTRING NOT ACCEPTED\n");
   return 0;}
```

2.Construction DFA:

```
#include <stdio.h>
int main() {
   FILE *fp = fopen("dfa.txt", "r");
   int Fa[10][10], states[3][10], row = 0, col = 0, sr = 0, sc = 0, in, curr, i, j;
   char str[100], k;
   int flag = 0;
   if (!fp) {
      printf("File could not be found\n");
      return 1; }
   for (i = 0; i < 3; i++)
      for (j = 0; j < 10; j++)
         states[i][j] = -1;
   while (fscanf(fp, "%d", &in) != EOF) {
      k = fgetc(fp);
      if (flag)
         states[sr][sc++] = in;
      else
         Fa[row][col++] = in;
      if (k == '\n') {
         if (flag) sr++, sc = 0;
         else row++, col = 0;
      } else if (k == '#') {
         flag = 1;
      } }
   fclose(fp);
   while (1) {
      printf("\nEnter the string: ");
      scanf("%s", str);
      curr = states[0][0];
      printf("%d", curr);
      for (i = 0; str[i] != '\0'; i++) {
         int next = Fa[curr][str[i] - 'a'];
         printf(" %c %d", str[i], next);
         if (next == -1) break;
         curr = next;}
      flag = 0;
      for (i = 0; i < 10 && states[1][i] != -1; i++) {
         if (curr == states[1][i]) {
            flag = 1;
            break;}}
      printf("\nSTRING %s\n", flag ? "ACCEPTED" : "NOT ACCEPTED");    }
   return 0;}
```

3.Generation of Tokens for Given Lexeme

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int isKeyword(char *s) {
   char *kw[] = {"if","else","while","do","for","int","float","return","char"};
   for (int i = 0; i < 9; i++)
      if (!strcmp(s, kw[i])) return 1;
   return 0; }
int main() {
   FILE *fp = fopen("source.txt", "r");
   char ch, str[20], ops[] = "+-*/%=", i = 0;
   if (!fp) return printf("File error\n"), 1;
   while ((ch = fgetc(fp)) != EOF) {
      if (strchr(ops, ch)) printf("Operator: %c\n", ch);
      else if (isalnum(ch)) str[i++] = ch;
      else if ((ch == ' ' || ch == '\n' || ch == ';' || ch == ',') && i) {
         str[i] = '\0'; i = 0;
         printf("%s: %s\n", isKeyword(str) ? "Keyword" : "Identifier", str); }}
   fclose(fp);
   return 0; }
```

4.Generation of token for given lexeme:

```
#include <stdio.h> #include <string.h> #include <ctype.h> #define MAX_LEN 100
const char *keywords[] = {"int", "float", "if", "else", "return", "while", "for", "char", "double", "void",
"main"};
const char delimiters[] = "(){};"; const char operators[] = "+-*/<>=!&|";
int isKeyword(char *word) {
   for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)
      if (strcmp(word, keywords[i]) == 0) return 1;   return 0;}
int isOperator(char ch) { return strchr(operators, ch) != NULL; }
int isDelimiter(char ch) { return strchr(delimiters, ch) != NULL; }
int isNumber(char *str) {
   for (int i = 0; str[i] != '\0'; i++)
      if (!isdigit(str[i]) && str[i] != '.') return 0;   return 1;}
int main() {
   char input[MAX_LEN], word[MAX_LEN];
   printf("Enter the string: ");
   fgets(input, MAX_LEN, stdin);
   int i = 0, j = 0;
   char delimitersFound[MAX_LEN] = "", operatorsFound[MAX_LEN] = "",
identifiers[MAX_LEN][MAX_LEN], keywordsFound[MAX_LEN][MAX_LEN],
literals[MAX_LEN][MAX_LEN], constants[MAX_LEN][MAX_LEN];
   int idCount = 0, kwCount = 0, litCount = 0, constCount = 0;
   while (input[i] != '\0') {char ch = input[i]; if (ch == '\"') { j = 0;
         i++;
         while (input[i] != '\"' && input[i] != '\0') word[j++] = input[i++];
         word[j] = '\0';
         if (input[i] == '\"') i++;
         strcpy(literals[litCount++], word);
      } else if (isOperator(ch)) {
         strncat(operatorsFound, &ch, 1);
      } else if (isDelimiter(ch)) {
         strncat(delimitersFound, &ch, 1);
      } else if (isalnum(ch) || ch == '.') { j = 0;
         while (isalnum(input[i]) || input[i] == '.') word[j++] = input[i++];
         word[j] = '\0';  i--;
         if (isKeyword(word)) strcpy(keywordsFound[kwCount++], word);
         else if (isNumber(word)) strcpy(constants[constCount++], word);
         else strcpy(identifiers[idCount++], word);} i++;}
   printf("\nDelimiters: %s\nOperators: %s\nIdentifiers: ", delimitersFound, operatorsFound);
   for (int k = 0; k < idCount; k++) printf("%s ", identifiers[k]);
   printf("\nKeywords: ");
   for (int k = 0; k < kwCount; k++) printf("%s ", keywordsFound[k]);
   printf("\nConstants: ");
   for (int k = 0; k < constCount; k++) printf("%s ", constants[k]);
   printf("\nLiterals: ");
   for (int k = 0; k < litCount; k++) printf("\"%s\" ", literals[k]);  return 0;}
```

5.Conversion of infix to postfix

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
typedef struct {
   int top;
   char items[MAX];
} Stack;
void initStack(Stack *s) { s->top = -1; }
int isEmpty(Stack *s) { return s->top == -1; }
void push(Stack *s, char c) { s->items[++(s->top)] = c; }
char pop(Stack *s) { return isEmpty(s) ? 0 : s->items[(s->top)--]; }
int precedence(char c) { return (c == '+' || c == '-') ? 1 : (c == '*' || c == '/') ? 2 : (c == '^') ? 3 :
0; }
int isOperator(char c) { return strchr("+-*/^", c) != NULL; }
void infixToPostfix(char *infix, char *postfix) {
   Stack s; initStack(&s);
   int i, j = 0;
   for (i = 0; infix[i]; i++) {
      char token = infix[i];
      if (isspace(token)) continue;
      if (isalnum(token)) postfix[j++] = token;
      else if (token == '(') push(&s, token);
      else if (token == ')') {
         while (!isEmpty(&s) && pop(&s) != '(') postfix[j++] = pop(&s);
      } else if (isOperator(token)) {
         while (!isEmpty(&s) && precedence(s.items[s.top]) >= precedence(token)) postfix[j++]
= pop(&s);
         push(&s, token);}
   } while (!isEmpty(&s)) postfix[j++] = pop(&s);   postfix[j] = '\0'; }
int main() {
   char infix[MAX], postfix[MAX];
   printf("Enter the expression: ");
   fgets(infix, MAX, stdin);
   infix[strcspn(infix, "\n")] = 0;
   infixToPostfix(infix, postfix);
   printf("Postfix Expression: %s\n", postfix); return 0;}
```

6.Implementtion for symbol table :

```
#include <stdio.h>
#include <string.h>
#define MAX 10
typedef struct {
   char name[20];
   char type[10];
   int size;
} Symbol;
Symbol table[MAX];
int count = 0;
void insert(char* name, char* type, int size) {
   if (count >= MAX) {
      printf("Symbol Table Full!\n");
      return;}
   strcpy(table[count].name, name);
   strcpy(table[count].type, type);
   table[count].size = size;
   count++; }
void search(char* name) {
   for (int i = 0; i < count; i++) {
      if (strcmp(table[i].name, name) == 0) {
         printf("Found: %s (%s, Size: %d)\n", table[i].name, table[i].type, table[i].size);
         return;}}
   printf("Symbol not found.\n")}
void display() {
   printf("\nSymbol Table:\n-------------------------\n");
   for (int i = 0; i < count; i++)
      printf("%s\t%s\t%d\n", table[i].name, table[i].type, table[i].size);
   printf("-------------------------\n");}
int main() {
   insert("x", "int", 4);
   insert("y", "float", 4);
   insert("arr", "int[10]", 40);
   display();
   printf("\nSearch symbol: ");
   char key[20];
   scanf("%19s", key);  // Fixed potential buffer overflow
   search(key);
   return 0;}
```

7.Syntax Analysis using YACC:

A.Lex File:

```
%{
#include "y.tab.h"
%}

%%
[0-9]+        { yylval = atoi(yytext); return NUMBER; }
[\t ]+        { /* Ignore whitespace */ }
\n            { return '\n'; }
.             { return yytext[0]; }
%%
```

B.YACC File

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s) { printf("Error: %s\n", s); }
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right '^'

%%
expr: expr '+' expr   { printf("Add\n"); }
   | expr '-' expr   { printf("Subtract\n"); }
   | expr '*' expr   { printf("Multiply\n"); }
   | expr '/' expr   { printf("Divide\n"); }
   | '(' expr ')'    { }
   | NUMBER          { printf("Number: %d\n", $1); }
   ;
%%
int main() {
   printf("Enter expression:\n");
   yyparse();
   return 0;
}
```

Step to run:

```
bison -d expr.y     # Generates y.tab.c and y.tab.h
flex expr.l         # Generates lex.yy.c
gcc lex.yy.c y.tab.c -o expr -ll
./expr
```

8.Implementaiton of Shift reduce parsing: #include <stdio.h>

```
#include <string.h>
#define MAX 100
char input[MAX], stack[MAX];
int top = -1, ip = 0;
void push(char c) { stack[++top] = c; }
void display() {
   printf("$");
   for (int i = 0; i <= top; i++) printf("%c", stack[i]); }
void reduce() {
   while (1) {
      if (top >= 0 && (stack[top] == 'a' || stack[top] == 'b')) {
         stack[top] = 'E'; printf("\t\tE->%c\n", stack[top]);
      } else if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] == 'E') {
         top -= 2; printf("\t\tE->E+E\n");
      } else if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '*' && stack[top - 2] == 'E') {
         top -= 2; printf("\t\tE->E*E\n");
      } else break;}}
int main() {
   printf("SHIFT REDUCE PARSER\n\nGRAMMAR:\nE -> E+E\nE -> E*E\nE -> a\nE -> b\n\n");
   printf("Enter input: ");
   scanf("%s", input);
   printf("\nStack\t\tInput\t\tAction\n---------------------------------------\n");
   while (input[ip] != '\0') {
      push(input[ip]);
      display(); printf("\t\t%s\t\tShift %c\n", input + ip + 1, input[ip]);
      ip++;
      reduce();}
   reduce();
   printf("--------------------\n");
   if (top == 0 && stack[top] == 'E')
      printf("ACCEPTED\n");
   else
      printf("PARSING FAILED\n");
   return 0;}
```