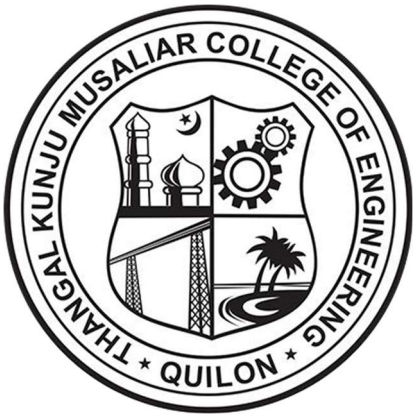
Thangal Kunju Musaliar College of Engineering

Kollam, Kerala



CSL 411 - COMPILER LAB

Department of Computer Science and Engineering

DECEMBER 2024

Thangal Kunju Musaliar College of Engineering

Kollam, Kerala

CSL 411 - COMPILER LAB RECORD

CERTIFICATE

Certified that this is a bonafide record of the work done by … ..... of …….. Semester, Class Roll No .....in the…… Laboratory during the year ……

Reg.no : ……..

Name of the Examination: **……..**

Staff member in charge *External Examiner*

VISION

To be a centre of excellence imparting quality education in Computer Science and Engineering and transforming students to critical thinkers and lifelong learners capable of developing environment friendly and economically feasible solutions to real world problems.

MISSION

- To provide strong foundation in Computer Science and Engineering, prepare students for professional career and higher education, and inculcate research interest.
- To be abreast of the technological advances in a rapidly changing world.
- To impart skills to come up with socially acceptable solutions to real world problems, upholding ethical values.

PROGRAMME EDUCATIONAL OBJECTIVES(PEOS)

PEO 1: Excel in professional career by acquiring knowledge in mathematics, science, and engineering and applying the knowledge in the design of hardware and software Solutions for challenging problems of the society.

PEO 2: Pursue higher studies and research thereby engages in lifelong learning by adapting to the current trends in the area of Computer Science & Engineering.

PEO 3: Ability to provide socially acceptable and economically feasible computer oriented solutions to real world problems with teamwork, while maintaining environmental balance, quality and cognizance of the underlying principles of ethics.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: Apply mathematical and algorithmic principles, data structure concepts, software, and hardware and techniques in designing and developing optimized and secure computer based solutions.

PSO2: Design and develop system software and provide exposure to various tools and programming languages to facilitate efficient computing environment which adds to the ease of human life.

PSO3: Use the knowledge of various data processing, communication and intelligent systems to provide solutions to new ideas and innovations.

PROGRAMME OUTCOMES

1. Engineering knowledge : Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis : Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first Principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions : Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems : Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society : Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability : Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics : Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work : Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication : Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance : Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning : Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOMES

CO1	Implement lexical analyzer to convert the input language to tokens
CO2	Design NFA and DFA for a problem and write programs to perform operations on it
CO3	Design and Implement Top-Down and Bottom-Up parsers
CO4	Implement intermediate code for expressions.
CO5	Implement code optimization techniques and generate assembly language for the optimized intermediate expressions

INDEX

SI NO.	EXPERIMENT NAME	PAGE NO	DATE
CYCLE 1			
1	LEXICAL ANALYSER		
2	EPSILON CLOSURE		
3	EPSILON NFA TO NFA		
4	NFA TO DFA		
5	DFA MINIMIZATION		
CYCLE 2			
1	LEX ANALYSER USING LEX		
2	FREQUENCY OF LINES, WORDS AND CHARACTERS		
3	VALID ARITHEMETIC EXPRESSION		
4	VALID IDENTIFIER		
CYCLE 3			
1	FIRST AND FOLLOW		
2	RECURSIVE DESCENT PARSER		
3	SHIFT REDUCE PARSER		
4	CONSTANT PROPOGATION		
5	INTERMEDIATE CODE GENERATION		
6	ASSEMBLY CODE GENERATION		

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name: Lexical Analyser*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
//comments
bool keywordCheck(const char* str) {
   const char* keywords[] = {"int", "return", "float", "char", "if", "else", "while", "for", "do",
"void", "double", "break"};
   int numKeywords = sizeof(keywords) / sizeof(char*);

   for (int i = 0; i < numKeywords; i++) {
      if (strcmp(str, keywords[i]) == 0) {
         printf("<%s, KEYWORD> ", str);
         return true;
      }
   }
   return false;
}
bool operatorCheck(const char* str) {
   const char* operators[] = {"+", "-", "*", "/", "=", "==", "<", ">", "<=", ">=", "!="};
   int numOperators = sizeof(operators) / sizeof(char*);

   for (int i = 0; i < numOperators; i++) {
      if (strcmp(str, operators[i]) == 0) {
         printf("<%s, OPERATOR> ", str);
         return true;
      }
   }
   return false;
}
bool separatorCheck(const char* str) {
   const char* separators[] = {";", ",", "(", ")", "{", "}"};
   int numSeparators = sizeof(separators) / sizeof(char*);

   for (int i = 0; i < numSeparators; i++) {
      if (strcmp(str, separators[i]) == 0) {
         printf("<%s, SEPARATOR> ", str);
         return true;
      }
   }
   return false;
}
```

```
bool integerCheck(const char* str) {
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        if (!isdigit(str[i])) {
            return false;
        }
    }
    printf("<%s, INTEGER> ", str);
    return true;
}
bool realNumberCheck(const char* str) {
    bool hasDecimal = false;
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        if (str[i] == '.') {
            if (hasDecimal) return false;
            hasDecimal = true;
        } else if (!isdigit(str[i])) {
            return false;
        }
    }
    if (hasDecimal) {
        printf("<%s, REAL NUMBER> ", str);
        return true;
    }
    return false;
}
bool identifierCheck(const char* str) {
    if (!isalpha(str[0]) && str[0] != '_') {
        return false;
    }
    int len = strlen(str);
    for (int i = 1; i < len; i++) {
        if (!isalnum(str[i]) && str[i] != '_') {
            return false;
        }
    }
    printf("<%s, IDENTIFIER> ", str);
    return true;
}
bool isDelimiter(char c) {
    return isspace(c) || c == ';' || c == ',' || c == '(' || c == ')' || c == '{' || c == '}';
}
bool preprocessorCheck(const char* str) {
    if (str[0] == '#') {
        printf("<%s, PREPROCESSOR DIRECTIVE> ", str);
        return true;
```

```
    }
    return false;
}
// Function to check for single-line and multi-line comments
bool commentCheck(const char* str, int* inMultilineComment) {
    if (*inMultilineComment) {
        const char* endComment = strstr(str, "*/");
        if (endComment) {
            *inMultilineComment = 0;
        }
        return true;
    }
    if (strstr(str, "//")) {
        return true;
    }
    const char* startMultiLine = strstr(str, "/*");
    if (startMultiLine) {
        *inMultilineComment = 1;
        const char* endComment = strstr(startMultiLine, "*/");
        if (endComment) {
            *inMultilineComment = 0;
        }
        return true;
    }
    return false;
}
void parse(const char* str, int lineNumber, int* inMultilineComment) {
    if (strlen(str) == 0 || (strlen(str) == 1 && str[0] == '\n')) {
        return;
    }
    printf("Line %d: ", lineNumber);
    if (preprocessorCheck(str)) {
        printf("\n");
        return;
    }
    if (commentCheck(str, inMultilineComment)) {
        printf("<COMMENT> \n");
        return;
    }
    int len = strlen(str);
    int start = 0, end = 0;
    while (end < len) {
        while (isspace(str[start])) start++;
        end = start;
        if (isDelimiter(str[end])) {
            char token[2] = {str[end], '\0'};
            separatorCheck(token);
```

```
            start = ++end;
            continue;
        }
        while (end < len && !isDelimiter(str[end])) end++;
        char lexeme[100];
        strncpy(lexeme, &str[start], end - start);
        lexeme[end - start] = '\0';
        if (!keywordCheck(lexeme) &&
            !operatorCheck(lexeme) &&
            !integerCheck(lexeme) &&
            !realNumberCheck(lexeme)) {
            identifierCheck(lexeme);
        }
        start = end;
    }
    printf("\n");
}
int main() {
    FILE* filePointer;
    char str[100];
    filePointer = fopen("text.txt", "r");
    if (!filePointer) {
        printf("File failed to open.");
        return 0;
    }
    int lineNumber = 1;
    int inMultilineComment = 0;
    while (fgets(str, 100, filePointer)) {
        if (strlen(str) > 1 || (strlen(str) == 1 && str[0] != '\n')) {
            parse(str, lineNumber, &inMultilineComment);
        }
        lineNumber++;
    }
    fclose(filePointer);
    return 0;
}
```

OUTPUT

```
Line 1: <#include <stdio.h>
, PREPROCESSOR DIRECTIVE>
Line 2: <#include <string.h>
, PREPROCESSOR DIRECTIVE>
Line 3: <#include <ctype.h>
, PREPROCESSOR DIRECTIVE>
Line 4: <#include <stdbool.h>
, PREPROCESSOR DIRECTIVE>
Line 5: <COMMENT>
```

Line 6: <bool, IDENTIFIER> <keywordCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 7: <const, IDENTIFIER> <=, OPERATOR> <{, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR>
Line 8: <,, SEPARATOR> <,, SEPARATOR> <}, SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 9: <int, KEYWORD> <numKeywords, IDENTIFIER> <=, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <keywords, IDENTIFIER> <), SEPARATOR> </, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 10: <, INTEGER>
Line 11: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <numKeywords, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 12: <if, KEYWORD> <(, SEPARATOR> <strcmp, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <), SEPARATOR> <==, OPERATOR> <0, INTEGER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 13: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 14: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 15: <}, SEPARATOR> <, INTEGER>
Line 16: <}, SEPARATOR> <, INTEGER>
Line 17: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 18: <}, SEPARATOR> <, INTEGER>
Line 19: <bool, IDENTIFIER> <operatorCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 20: <const, IDENTIFIER> <=, OPERATOR> <{, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <}, SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 21: <int, KEYWORD> <numOperators, IDENTIFIER> <=, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <operators, IDENTIFIER> <), SEPARATOR> </, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 22: <, INTEGER>
Line 23: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <numOperators, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 24: <if, KEYWORD> <(, SEPARATOR> <strcmp, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <), SEPARATOR> <==, OPERATOR> <0, INTEGER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 25: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 26: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>

Line 27: <}, SEPARATOR> <, INTEGER>
Line 28: <}, SEPARATOR> <, INTEGER>
Line 29: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 30: <}, SEPARATOR> <, INTEGER>
Line 31: <bool, IDENTIFIER> <separatorCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 32: <const, IDENTIFIER> <=, OPERATOR> <{, SEPARATOR> <;, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <(, SEPARATOR> <,, SEPARATOR> <), SEPARATOR> <,, SEPARATOR> <{, SEPARATOR> <,, SEPARATOR> <}, SEPARATOR> <}, SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 33: <int, KEYWORD> <numSeparators, IDENTIFIER> <=, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <separators, IDENTIFIER> <), SEPARATOR> </, OPERATOR> <sizeof, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 34: <, INTEGER>
Line 35: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <numSeparators, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 36: <if, KEYWORD> <(, SEPARATOR> <strcmp, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <), SEPARATOR> <==, OPERATOR> <0, INTEGER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 37: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 38: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 39: <}, SEPARATOR> <, INTEGER>
Line 40: <}, SEPARATOR> <, INTEGER>
Line 41: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 42: <}, SEPARATOR> <, INTEGER>
Line 43: <bool, IDENTIFIER> <integerCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 44: <int, KEYWORD> <len, IDENTIFIER> <=, OPERATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 45: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <len, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 46: <if, KEYWORD> <(, SEPARATOR> <(, SEPARATOR> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 47: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 48: <}, SEPARATOR> <, INTEGER>
Line 49: <}, SEPARATOR> <, INTEGER>
Line 50: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 51: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 52: <}, SEPARATOR> <, INTEGER>
Line 53: <bool, IDENTIFIER> <realNumberCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>

Line 54: <bool, IDENTIFIER> <hasDecimal, IDENTIFIER> <=, OPERATOR> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 55: <int, KEYWORD> <len, IDENTIFIER> <=, OPERATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 56: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <len, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 57: <if, KEYWORD> <(, SEPARATOR> <==, OPERATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 58: <if, KEYWORD> <(, SEPARATOR> <hasDecimal, IDENTIFIER> <), SEPARATOR> <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 59: <hasDecimal, IDENTIFIER> <=, OPERATOR> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 60: <}, SEPARATOR> <else, KEYWORD> <if, KEYWORD> <(, SEPARATOR> <(, SEPARATOR> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 61: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 62: <}, SEPARATOR> <, INTEGER>
Line 63: <}, SEPARATOR> <, INTEGER>
Line 64: <if, KEYWORD> <(, SEPARATOR> <hasDecimal, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 65: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <REAL, IDENTIFIER> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 66: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 67: <}, SEPARATOR> <, INTEGER>
Line 68: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 69: <}, SEPARATOR> <, INTEGER>
Line 70: <bool, IDENTIFIER> <identifierCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 71: <if, KEYWORD> <(, SEPARATOR> <(, SEPARATOR> <), SEPARATOR> <!=, OPERATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 72: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 73: <}, SEPARATOR> <, INTEGER>
Line 74: <int, KEYWORD> <len, IDENTIFIER> <=, OPERATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 75: <for, KEYWORD> <(, SEPARATOR> <int, KEYWORD> <i, IDENTIFIER> <=, OPERATOR> <1, INTEGER> <;, SEPARATOR> <i, IDENTIFIER> <<, OPERATOR> <len, IDENTIFIER> <;, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 76: <if, KEYWORD> <(, SEPARATOR> <(, SEPARATOR> <), SEPARATOR> <!=, OPERATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 77: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 78: <}, SEPARATOR> <, INTEGER>
Line 79: <}, SEPARATOR> <, INTEGER>
Line 80: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 81: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 82: <}, SEPARATOR> <, INTEGER>

Line 83: <bool, IDENTIFIER> <isDelimiter, IDENTIFIER> <(, SEPARATOR> <char, KEYWORD> <c, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 84: <return, KEYWORD> <isspace, IDENTIFIER> <(, SEPARATOR> <c, IDENTIFIER> <), SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <;, SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <,, SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <(, SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <), SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <{, SEPARATOR> <c, IDENTIFIER> <==, OPERATOR> <}, SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 85: <}, SEPARATOR> <, INTEGER>
Line 86: <bool, IDENTIFIER> <preprocessorCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 87: <if, KEYWORD> <(, SEPARATOR> <==, OPERATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 88: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <PREPROCESSOR, IDENTIFIER> <,, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 89: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 90: <}, SEPARATOR> <, INTEGER>
Line 91: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 92: <}, SEPARATOR> <, INTEGER>
Line 93: <COMMENT>
Line 94: <bool, IDENTIFIER> <commentCheck, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <,, SEPARATOR> <inMultilineComment, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 95: <if, KEYWORD> <(, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 96: <const, IDENTIFIER> <endComment, IDENTIFIER> <=, OPERATOR> <strstr, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 97: <if, KEYWORD> <(, SEPARATOR> <endComment, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 98: <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 99: <}, SEPARATOR> <, INTEGER>
Line 100: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 101: <}, SEPARATOR> <, INTEGER>
Line 102: <COMMENT>
Line 103: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 104: <}, SEPARATOR> <, INTEGER>
Line 105: <COMMENT>
Line 106: <COMMENT>
Line 107: <COMMENT>
Line 108: <COMMENT>
Line 109: <if, KEYWORD> <(, SEPARATOR> <endComment, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 110: <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 111: <}, SEPARATOR> <, INTEGER>
Line 112: <return, KEYWORD> <true, IDENTIFIER> <;, SEPARATOR> <, INTEGER>

Line 113: <}, SEPARATOR> <, INTEGER>
Line 114: <return, KEYWORD> <false, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 115: <}, SEPARATOR> <, INTEGER>
Line 116: <void, KEYWORD> <parse, IDENTIFIER> <(, SEPARATOR> <const, IDENTIFIER> <str, IDENTIFIER> <,, SEPARATOR> <int, KEYWORD> <lineNumber, IDENTIFIER> <,, SEPARATOR> <inMultilineComment, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 117: <if, KEYWORD> <(, SEPARATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <==, OPERATOR> <0, INTEGER> <(, SEPARATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <==, OPERATOR> <1, INTEGER> <==, OPERATOR> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 118: <return, KEYWORD> <;, SEPARATOR> <, INTEGER>
Line 119: <}, SEPARATOR> <, INTEGER>
Line 120: <printf, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <lineNumber, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 121: <if, KEYWORD> <(, SEPARATOR> <preprocessorCheck, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 122: <printf, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 123: <return, KEYWORD> <;, SEPARATOR> <, INTEGER>
Line 124: <}, SEPARATOR> <, INTEGER>
Line 125: <if, KEYWORD> <(, SEPARATOR> <commentCheck, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <inMultilineComment, IDENTIFIER> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 126: <printf, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 127: <return, KEYWORD> <;, SEPARATOR> <, INTEGER>
Line 128: <}, SEPARATOR> <, INTEGER>
Line 129: <int, KEYWORD> <len, IDENTIFIER> <=, OPERATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 130: <int, KEYWORD> <start, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <,, SEPARATOR> <end, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 131: <while, KEYWORD> <(, SEPARATOR> <end, IDENTIFIER> <<, OPERATOR> <len, IDENTIFIER> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 132: <while, KEYWORD> <(, SEPARATOR> <isspace, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 133: <end, IDENTIFIER> <=, OPERATOR> <start, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 134: <if, KEYWORD> <(, SEPARATOR> <isDelimiter, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 135: <char, KEYWORD> <=, OPERATOR> <{, SEPARATOR> <,, SEPARATOR> <}, SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 136: <separatorCheck, IDENTIFIER> <(, SEPARATOR> <token, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>

Line 137: <start, IDENTIFIER> <=, OPERATOR> <;, SEPARATOR> <, INTEGER>
Line 138: <continue, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 139: <}, SEPARATOR> <, INTEGER>
Line 140: <while, KEYWORD> <(, SEPARATOR> <end, IDENTIFIER> <<, OPERATOR> <len, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 141: <char, KEYWORD> <;, SEPARATOR> <, INTEGER>
Line 142: <strncpy, IDENTIFIER> <(, SEPARATOR> <lexeme, IDENTIFIER> <,, SEPARATOR> <,, SEPARATOR> <end, IDENTIFIER> <-, OPERATOR> <start, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 143: <-, OPERATOR> <=, OPERATOR> <;, SEPARATOR> <, INTEGER>
Line 144: <if, KEYWORD> <(, SEPARATOR> <(, SEPARATOR> <lexeme, IDENTIFIER> <), SEPARATOR> <, INTEGER>
Line 145: <(, SEPARATOR> <lexeme, IDENTIFIER> <), SEPARATOR> <, INTEGER>
Line 146: <(, SEPARATOR> <lexeme, IDENTIFIER> <), SEPARATOR> <, INTEGER>
Line 147: <(, SEPARATOR> <lexeme, IDENTIFIER> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 148: <identifierCheck, IDENTIFIER> <(, SEPARATOR> <lexeme, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 149: <}, SEPARATOR> <, INTEGER>
Line 150: <start, IDENTIFIER> <=, OPERATOR> <end, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 151: <}, SEPARATOR> <, INTEGER>
Line 152: <printf, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 153: <}, SEPARATOR> <, INTEGER>
Line 154: <int, KEYWORD> <main, IDENTIFIER> <(, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 155: <filePointer, IDENTIFIER> <;, SEPARATOR> <, INTEGER>
Line 156: <char, KEYWORD> <;, SEPARATOR> <, INTEGER>
Line 157: <filePointer, IDENTIFIER> <=, OPERATOR> <fopen, IDENTIFIER> <(, SEPARATOR> <,, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 158: <if, KEYWORD> <(, SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 159: <printf, IDENTIFIER> <(, SEPARATOR> <failed, IDENTIFIER> <to, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 160: <return, KEYWORD> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 161: <}, SEPARATOR> <, INTEGER>
Line 162: <int, KEYWORD> <lineNumber, IDENTIFIER> <=, OPERATOR> <1, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 163: <int, KEYWORD> <inMultilineComment, IDENTIFIER> <=, OPERATOR> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 164: <while, KEYWORD> <(, SEPARATOR> <fgets, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <100, INTEGER> <,, SEPARATOR> <filePointer, IDENTIFIER> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 165: <if, KEYWORD> <(, SEPARATOR> <strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <>, OPERATOR> <1, INTEGER> <(, SEPARATOR>

<strlen, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <), SEPARATOR> <==, OPERATOR> <1, INTEGER> <!=, OPERATOR> <), SEPARATOR> <), SEPARATOR> <{, SEPARATOR> <, INTEGER>
Line 166: <parse, IDENTIFIER> <(, SEPARATOR> <str, IDENTIFIER> <,, SEPARATOR> <lineNumber, IDENTIFIER> <,, SEPARATOR> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 167: <}, SEPARATOR> <, INTEGER>
Line 168: <;, SEPARATOR> <, INTEGER>
Line 169: <}, SEPARATOR> <, INTEGER>
Line 170: <fclose, IDENTIFIER> <(, SEPARATOR> <filePointer, IDENTIFIER> <), SEPARATOR> <;, SEPARATOR> <, INTEGER>
Line 171: <return, KEYWORD> <0, INTEGER> <;, SEPARATOR> <, INTEGER>
Line 172: <}, SEPARATOR> <, INTEGER>

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name: E Closure of all states of an NFA with E transition*/

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

typedef struct trans {
	char state1[3];
	char input[3];
	char state2[3];
} trans;

char states[10][3], result[10][3], temp[3];

int add_state(char state[3], int k) {
	int flag = 0;

	for(int i = 0; i < k; i++) {
		if(!strcmp(result[i], state)) flag = 1;
	}

	if(!flag) {
		strcpy(result[k], state);
		k++;
	}

	return k;
}

void display(int n){
	printf("Epsilon closure of %s = {", temp);

	for(int i = 0; i < n; i++)
		printf(" %s", result[i]);

	printf(" }\n");
}

void main() {
	int k, n, m;
	char state[3];

	printf("Enter the no of states: ");
	scanf("%d", &n);

	printf("Enter the states:\n");
	for(int i = 0; i < n; i++){
		scanf("%s", states[i]);
	}
```

```
        printf("\nEnter the number of transitions: ");
        scanf("%d", &m);

        trans table[m];

        printf("Enter the transition table:\n");
        for(int i = 0; i < m; i++) {
                scanf("%s %s %s", table[i].state1, table[i].input, table[i].state2);
        }

        printf("\n");
        for(int i = 0; i < n; i++){
                k = 0;

                strcpy(temp, states[i]);
                k = add_state(states[i], k);

                for(int p = 0; p < k; p++) {
                        strcpy(state, result[p]);

                        for(int j = 0; j < m; j++){
                                if(!strcmp(state, table[j].state1)){
                                        if(!strcmp(table[j].input, "e")) {
                                                k = add_state(table[j].state2, k);
                                        }
                                }
                        }
                }

                display(k);
        }
}
```

OUTPUT

```
Enter the no of states: 3
Enter the states:
q0
q1
q2

Enter the number of transitions: 5
Enter the transition table:
q0 a q0
q0 e q1
q1 b q1
q1 e q2
q2 c q2

Epsilon closure of q0 = { q0 q1 q2 }
Epsilon closure of q1 = { q1 q2 }
Epsilon closure of q2 = { q2 }
```

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name: E-NFA TO NFA*/

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

typedef struct trans {
	char state1[3];
	char input[3];
	char state2[3];
} trans;

typedef struct closure {
	int len;
	char states[10][3];
} closure;

char states[10][3], temp[3], start[3], final[10][3];
closure cl[10];

int add_state(char state[3], int i, int k) {
	int flag = 0;

	for(int j = 0; j < k; j++) {
		if(!strcmp(cl[i].states[j], state)) flag = 1;
	}

	if(!flag) {
		strcpy(cl[i].states[k], state);
		k++;
		cl[i].len = k;
	}

	return k;
}

int isDup(trans table[], int len, char state1[], char input[], char state2[]) {
	int flag = 0;

	for(int i = 0; i < len; i++) {
		if(!strcmp(table[i].state1, state1) && !strcmp(table[i].input, input)
&& !strcmp(table[i].state2, state2))
			flag = 1;
	}

	return flag;
}

void main() {
	int n, m, f;
```

```
char state[3];

// Reading the e-nfa
printf("Enter the no of states: ");
scanf("%d", &n);

printf("Enter the states:\n");
for(int i = 0; i < n; i++){
        scanf("%s", states[i]);
}

printf("Enter the start state: ");
scanf("%s", start);

printf("Enter the number of final states: ");
scanf("%d", &f);

printf("Enter the final states\n");
for(int i = 0; i < f; i++) {
        scanf("%s", final[i]);
}

printf("Enter the number of transitions: ");
scanf("%d", &m);

trans table[m], table1[m * m];

printf("Enter the transition table:\n");
for(int i = 0; i < m; i++) {
        scanf("%s %s %s", table[i].state1, table[i].input, table[i].state2);
}

int l;

// Finding e-closure of all states
for(int i = 0; i < n; i++){
        l = 0;

        strcpy(temp, states[i]);
        l = add_state(states[i], i, l);

        for(int p = 0; p < l; p++) {
                strcpy(state, cl[i].states[p]);

                for(int j = 0; j < m; j++){
                        if(!strcmp(state, table[j].state1)){
                                if(!strcmp(table[j].input, "e")) {
                                        l = add_state(table[j].state2, i, l);
                                }
                        }
                }
        }
```

```
}

// Finding e-closure
printf("\nE-closure\n");
for(int i = 0; i < n; i++) {
	printf("%s: ", states[i]);
	for(int j = 0; j < cl[i].len; j++) {
		printf(" %s ", cl[i].states[j]);
	}
	printf("\n");
}

int s1 = 0, m1 = 0, f1 = 0;
char start1[10][3];
char final1[10][3];

// Finding new start states
for(int i = 0; i < n; i++) {
	if(!strcmp(states[i], start)) {
		for(int j = 0; j < cl[i].len; j++) {
			strcpy(start1[s1], cl[i].states[j]);
			s1++;
		}

		break;
	}
}

printf("\nThe start states of new NFA:\n");
for(int i = 0; i < s1; i++) {
	printf("%s\n", start1[i]);
}

// Finding new final states
for(int k = 0; k < f; k++) {
	for(int i = 0; i < n; i++) {
		for(int j = 0; j < cl[i].len; j++) {
			if(!strcmp(final[k], cl[i].states[j])) {
				strcpy(final1[f1], states[i]);
				f1++;
			}
		}
	}
}

printf("\nThe final states of new NFA:\n");
for(int i = 0; i < f1; i++) {
	printf("%s\n", final1[i]);
}

// Finding the transition table
for(int i = 0; i < n; i++) {
```

```
                for(int j = 0; j < cl[i].len; j++) {
                        for(int k = 0; k < m; k++) {
                                if(!strcmp(table[k].state1, cl[i].states[j]) && strcmp(table[k].input,
"e")) {
                                        for(int p = 0; p < n; p++) {
                                                if(!strcmp(states[p], table[k].state2)) {
                                                        for(int q = 0; q < cl[p].len; q++) {
                                                                if(!isDup(table1, m1, states[i],
table[k].input, cl[p].states[q])) {
                                                                        strcpy(table1[m1].state1,
states[i]);
                                                                        strcpy(table1[m1].input,
table[k].input);
                                                                        strcpy(table1[m1].state2,
cl[p].states[q]);
                                                                        m1++;
                                                                }
                                                        }

                                                        break;
                                                }
                                        }
                                }
                        }
                }
        }

        printf("\nTransition table of new NFA:\n");
        for(int i = 0; i < m1; i++) {
                printf("%s %s %s\n", table1[i].state1, table1[i].input, table1[i].state2);
        }
}
```

OUTPUT

```
Enter the no of states: 5
Enter the states:
q0
q1
q2
q3
q4
Enter the start state: q0
Enter the number of final states: 1
Enter the final states
q2
Enter the number of transitions: 7
Enter the transition table:
q0 1 q1
q1 1 q0
q0 e q2
q2 0 q3
```

q3 0 q2
q2 1 q4
q4 0 q2

E-closure
q0: q0 q2
q1: q1
q2: q2
q3: q3
q4: q4

The start states of new NFA:
q0
q2

The final states of new NFA:
q0
q2

Transition table of new NFA:
q0 1 q1
q0 0 q3
q0 1 q4
q1 1 q0
q1 1 q2
q2 0 q3
q2 1 q4
q3 0 q2
q4 0 q2

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name: NFA TO DFA*/

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

typedef struct trans {
	char state1[3];
	char input;
	char state2[3];
} trans;

typedef struct state {
	int len;
	char states[10][3];
	int flag;
} state;

typedef struct trans1 {
	state state1;
	char input;
	state state2;
} trans1;

// NFA data
char states[10][3], start[3], final[10][3], alpha[5];
trans table[30];

// DFA data
state states1[100], start1, final1[100];
trans1 table1[300];

void cpyState(state* state1, state state2) {
	state1->len = state2.len;
	state1->flag = state2.flag;

	for(int i = 0; i < state2.len; i++) {
		strcpy(state1->states[i], state2.states[i]);
	}
}

int cmpState(state state1, state state2) {
	if(state1.len != state2.len)
		return 0;

	int count = 0;

	for(int i = 0; i < state1.len; i++) {
		for(int j = 0; j < state2.len; j++) {
			if(!strcmp(state1.states[i], state2.states[j]))
```

```
                                count++;
                        }
                }

                if(count == state1.len)
                        return 1;
                else
                        return 0;
}

int containsState(state st, char state[]) {
        for(int i = 0; i < st.len; i++) {
                if(!strcmp(st.states[i], state))
                        return 1;
        }

        return 0;
}

void insertTrans(trans1* table, state* state1, char input, state* state2, int i) {
        table[i].state1.len = state1->len;
        table[i].state1.flag = state1->flag;

        for(int j = 0; j < state1->len; j++) {
                strcpy(table[i].state1.states[j], state1->states[j]);
        }

        table[i].state2.len = state2->len;
        table[i].state2.flag = state2->flag;

        for(int j = 0; j < state2->len; j++) {
                strcpy(table[i].state2.states[j], state2->states[j]);
        }

        table[i].input = input;
}

void main() {
        int n, f, m, l;

        // Reading the NFA
        printf("Enter the no of states: ");
        scanf("%d", &n);

        printf("Enter the states:\n");
        for(int i = 0; i < n; i++) {
                scanf("%s", states[i]);
        }

        printf("Enter the start state: ");
        scanf("%s", start);
```

```
	printf("Enter the number of final states: ");
	scanf("%d", &f);

	printf("Enter the final states\n");
	for(int i = 0; i < f; i++) {
		scanf("%s", final[i]);
	}

	printf("Enter the number of alphabets: ");
	scanf("%d", &l);

	printf("Enter the alphabets\n");
	for(int i = 0; i < l; i++) {
		scanf(" %c", &alpha[i]);
	}

	printf("Enter the number of transitions: ");
	scanf("%d", &m);

	printf("Enter the transition table:\n");
	for(int i = 0; i < m; i++) {
		scanf("%s %c %s", table[i].state1, &table[i].input, table[i].state2);
	}

	// Converting to DFA
	int n1 = 0, f1 = 0, m1 = 0;

	strcpy(start1.states[0], start); // Start state of DFA
	start1.len = 1;
	start1.flag = 1;

	cpyState(&states1[n1], start1); // Adding states
	n1++;

	state temp;

	cpyState(&temp, start1);

	int flag, isDead = 0;
	state dead;

	while(1) {
		for(int i = 0; i < l; i++) {
			state temp1;
			temp1.len = 0;
			temp1.flag = 0;

			for(int j = 0; j < temp.len; j++) {
				for(int k = 0; k < m; k++) {
					if(!strcmp(temp.states[j], table[k].state1) && alpha[i] ==
table[k].input && !containsState(temp1, table[k].state2)) {
						strcpy(temp1.states[temp1.len++], table[k].state2);
```

```
                        }
                    }
                }

                state state1, state2;

                cpyState(&state1, temp);

                if(temp1.len == 0) {
                    isDead = 1;
                    dead.len = 1;
                    dead.flag = 1;
                    strcpy(dead.states[0], "q-");
                    cpyState(&state2, dead);
                } else {
                    cpyState(&state2, temp1);

                    flag = 0;

                    for(int k = 0; k < n1; k++) {
                        if(cmpState(states1[k], temp1))
                            flag = 1;
                    }

                    if(!flag)
                        cpyState(&states1[n1++], temp1);
                }

                insertTrans(table1, &state1, alpha[i], &state2, m1++);
            }

            flag = 0;

            for(int i = 0; i < n1; i++) {
                if(states1[i].flag == 0) {
                    flag = 1;
                    cpyState(&temp, states1[i]);
                    states1[i].flag = 1;
                    break;
                }
            }

            if(!flag) {
                if(isDead) {
                    for(int i = 0; i < l; i++)
                        insertTrans(table1, &dead, alpha[i], &dead, m1++);
                }
                break;
            }
        }

        for(int i = 0; i < f; i++) {
```

```
            for(int j = 0; j < n1; j++) {
                for(int k = 0; k < states1[j].len; k++) {
                    if(!strcmp(final[i], states1[j].states[k])) {
                        cpyState(&final1[f1++], states1[j]);
                        break;
                    }
                }
            }
        }

        printf("\nStates of DFA\n");
        for(int i = 0; i < n1; i++) {
            printf("{");
            for(int j = 0; j < states1[i].len; j++) {
                printf(" %s ", states1[i].states[j]);
            }
            printf("}\n");
        }

        printf("\nStart state of DFA\n");
        printf("{");
        printf(" %s ", start1.states[0]);
        printf("}\n");

        printf("\nFinal states of DFA\n");
        for(int i = 0; i < f1; i++) {
            printf("{");
            for(int j = 0; j < final1[i].len; j++) {
                printf(" %s ", final1[i].states[j]);
            }
            printf("}\n");
        }


        printf("\nTransition table of DFA\n");
        for(int i = 0; i < m1; i++) {
            printf("{");
            for(int j = 0; j < table1[i].state1.len; j++) {
                printf(" %s ", table1[i].state1.states[j]);
            }
            printf("}");

            printf(" %c ", table1[i].input);

            printf("{");
            for(int j = 0; j < table1[i].state2.len; j++) {
                printf(" %s ", table1[i].state2.states[j]);
            }
            printf("}\n");
        }
}
```

OUTPUT

```
Enter the no of states: 3
Enter the states:
q0
q1
q2
Enter the start state: q0
Enter the number of final states: 1
Enter the final states
q2
Enter the number of alphabets: 2
Enter the alphabets
0
1
Enter the number of transitions: 9
Enter the transition table:
q0 0 q0
q0 1 q1
q0 1 q2
q1 0 q1
q1 0 q2
q1 1 q2
q2 0 q0
q2 0 q1
q2 1 q1

States of DFA
{ q0 }
{ q1  q2 }
{ q1  q2  q0 }

Start state of DFA
{ q0 }

Final states of DFA
{ q1  q2 }
{ q1  q2  q0 }

Transition table of DFA
{ q0 } 0 { q0 }
{ q0 } 1 { q1  q2 }
{ q1  q2 } 0 { q1  q2  q0 }
{ q1  q2 } 1 { q2  q1 }
{ q1  q2  q0 } 0 { q1  q2  q0 }
{ q1  q2  q0 } 1 { q2  q1 }
```

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name : Minimisation of DFA*/
#include <stdio.h>
#include <string.h>

typedef struct trans {
	char state1[3];
	char input;
	char state2[3];
} trans;

typedef struct state {
	int len;
	char states[10][3];
} state;

typedef struct trans1 {
	state state1;
	char input;
	state state2;
} trans1;

int store[10];

// DFA data
int n, f, m, l;
char states[10][3], start[3], final[10][3], alpha[5];
trans table[50];
int n1, f1, m1, l;
state states1[10], start1, final1[10];
trans1 table1[50];

void removeState(int k) {
	for(int i = k; i < n - 1; i++) {
		strcpy(states[i], states[i + 1]);
	}

	n--;
}

void removeTrans(int k) {
	for(int i = 2 * k; i < m - 2; i++) {
		strcpy(table[i].state1, table[i + 2].state1);
		table[i].input = table[i + 2].input;
		strcpy(table[i].state2, table[i + 2].state2);
	}

	m -= 2;
}
```

```
int isFinal(char state[]) {
	for(int i = 0; i < f; i++) {
		if(!strcmp(final[i], state))
			return 1;
	}

	return 0;
}

int goTo(char st[], char c) {
	for(int i = 0; i < m; i++) {
		if(!strcmp(table[i].state1, st) && table[i].input == c) {
			for(int j = 0; j < n; j++) {
				if(!strcmp(states[j], table[i].state2))
					return j;
			}
		}
	}
}

int contains(state st[], int n, char state[3]) {
	for(int i = 0; i < n; i++) {
		for(int j = 0; j < st[i].len; j++) {
			if(!strcmp(st[i].states[j], state))
				return 1;
		}
	}

	return 0;
}

int cmpState(state state1, state state2) {
	if(state1.len != state2.len)
		return 0;

	int count = 0;

	for(int i = 0; i < state1.len; i++) {
		for(int j = 0; j < state2.len; j++) {
			if(!strcmp(state1.states[i], state2.states[j]))
				count++;
		}
	}

	if(count == state1.len)
		return 1;
	else
		return 0;
}

int containsState(state states[], int n, state st) {
	for(int i = 0; i < n; i++) {
```

```
                if(cmpState(states[i], st))
                        return 1;
        }

        return 0;
}

int is_clique(int n, int graph[n][n], int b) {
        for(int i = 1; i < b; i++) {
                for(int j = i + 1; j < b; j++)
                        if(graph[store[i]][store[j]] == 1)
                                return 0;
        }

        return 1;
}

void combine(int n)
{
        states1[n1].len = 0;

        int flag = 0;

        for(int i = 1; i < n; i++) {
                if(!contains(states1, n1, states[store[i]])) {
                        strcpy(states1[n1].states[states1[n1].len++], states[store[i]]);
                        flag = 1;
                }
        }

        if(flag)
                n1++;
}

void findCliques(int n, int graph[n][n], int i, int l, int k) {
        for (int j = i; j < n - (k - l); j++) {
                store[l] = j;

                if (is_clique(n, graph, l + 1)) {
                        if (l < k)
                                findCliques(n, graph, j + 1, l + 1, k);
                        else
                                combine(l + 1);
                }
        }
}

void cpyState(state* state1, state state2) {
        state1->len = state2.len;

        for(int i = 0; i < state2.len; i++) {
                strcpy(state1->states[i], state2.states[i]);
```

```
        }
}

void insertTrans(trans1* table, state state1, char input, state state2, int i) {
        table[i].state1.len = state1.len;

        for(int j = 0; j < state1.len; j++) {
                strcpy(table[i].state1.states[j], state1.states[j]);
        }

        table[i].state2.len = state2.len;

        for(int j = 0; j < state2.len; j++) {
                strcpy(table[i].state2.states[j], state2.states[j]);
        }

        table[i].input = input;
}

void main() {
        printf("Enter the no of states: ");
        scanf("%d", &n);

        printf("Enter the states:\n");
        for(int i = 0; i < n; i++) {
                scanf("%s", states[i]);
        }

        printf("Enter the start state: ");
        scanf("%s", start);

        printf("Enter the number of final states: ");
        scanf("%d", &f);

        printf("Enter the final states\n");
        for(int i = 0; i < f; i++) {
                scanf("%s", final[i]);
        }

        printf("Enter the number of alphabets: ");
        scanf("%d", &l);

        printf("Enter the alphabets\n");
        for(int i = 0; i < l; i++) {
                scanf(" %c", &alpha[i]);
        }

        m = 0;
        printf("Enter the transitions\n");
        for(int i = 0; i < n; i++) {
                for(int j = 0; j < l; j++) {
                        strcpy(table[m].state1, states[i]);
```

```
                table[m].input = alpha[j];
                printf("%s %c ", states[i], alpha[j]);
                scanf("%s", table[m].state2);
                m++;
            }
        }

        // Minimising using Myhill-Nerode Theorem
        printf("\nMinimised DFA\n");

        int flag;

        // Eliminating unreachable states
        do {
            flag = 0;

            for(int i = 0; i < n; i++) {
                if(!strcmp(states[i], start))
                    continue;

                int flag1 = 0;

                for(int j = 0; j < m; j++) {
                    if(strcmp(table[j].state1, states[i]) && !strcmp(table[j].state2,
states[i]))
                        flag1 = 1;
                }

                if(!flag1) {
                    flag = 1;
                    removeState(i);
                    removeTrans(i);
                }
            }
        } while(flag);

        // Table filling
        int mTbl[n][n];

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                mTbl[i][j] = 0;
            }
        }

        for(int i = 0; i < n; i++) {
            for(int j = i + 1; j < n; j++) {
                if((isFinal(states[i]) && !isFinal(states[j])) || (!isFinal(states[i]) &&
isFinal(states[j])) ) {
                    mTbl[i][j] = 1;
                    mTbl[j][i] = 1;
                }
```

```
		}
	}

	int s1, s2;
		do {
		flag = 0;

		for(int i = 0; i < n; i++) {
			for(int j = i + 1; j < n; j++) {
				if(mTbl[i][j])
					continue;

				for(int k = 0; k < l; k++) {
					s1 = goTo(states[i], alpha[k]);
					s2 = goTo(states[j], alpha[k]);

					if(mTbl[s1][s2] || mTbl[s2][s1]) {
						mTbl[i][j] = 1;
						mTbl[j][i] = 1;
						flag = 1;
					}
				}
			}
		}
	} while(flag);

	// Combining states
	n1 = 0;

	for(int k = n; k > 0; k--) {
		findCliques(n, mTbl, 0, 1, k);
	}

	printf("\nStates:\n");
	for(int i = 0; i < n1; i++) {
		printf("{");
		for(int j = 0; j < states1[i].len; j++) {
			printf(" %s ", states1[i].states[j]);
		}
		printf("}\n");
	}

	// Transition table of min DFA
	m1 = 0;

	for(int i = 0; i < n1; i++) {
		for(int j = 0; j < l; j++) {
			for(int k = 0; k < m; k++) {
				if(!strcmp(table[k].state1, states1[i].states[0]) && table[k].input
== alpha[j]) {
					for(int i1 = 0; i1 < n1; i1++) {
						for(int j1 = 0; j1 < states1[i1].len; j1++) {
```

```
                                                if(!strcmp(states1[i1].states[j1],
table[k].state2)) {
                                                        insertTrans(table1, states1[i],
alpha[j], states1[i1], m1++);
                                                }
                                        }
                                }
                        }
                }
        }

        printf("\nTransition table:\n");
        for(int i = 0; i < m1; i++) {
                printf("{");
                for(int j = 0; j < table1[i].state1.len; j++) {
                        printf(" %s ", table1[i].state1.states[j]);
                }
                printf("}");

                printf(" %c ", table1[i].input);

                printf("{");
                for(int j = 0; j < table1[i].state2.len; j++) {
                        printf(" %s ", table1[i].state2.states[j]);
                }
                printf("}\n");
        }

        // Start state of min DFA
        for(int i = 0; i < n1; i++) {
                for(int j = 0; j < states1[i].len; j++) {
                        if(!strcmp(states1[i].states[j], start)) {
                                cpyState(&start1, states1[i]);
                        }
                }
        }

        printf("\nStart state: {");
        for(int i = 0; i < start1.len; i++) {
                printf(" %s ", start1.states[i]);
        }
        printf("}\n");

        // Final states of min DFA
        f1 = 0;
        for(int i = 0; i < n1; i++) {
                for(int j = 0; j < states1[i].len; j++) {
                        for(int k = 0; k < f; k++) {
                                if(!strcmp(states1[i].states[j], final[k]) && !containsState(final1,
f1, states1[i])) {
                                        cpyState(&final1[f1++], states1[i]);
```

```
                                }
                        }
                }
        }

        printf("\nFinal states:\n");
        for(int i = 0; i < f1; i++) {
                printf("{");
                for(int j = 0; j < final1[i].len; j++) {
                        printf(" %s ", final1[i].states[j]);
                }
                printf("}\n");
        }
}
```

OUTPUT

```
Enter the no of states: 4
Enter the states:
q0
q1
q2
q3
Enter the start state: q0
Enter the number of final states: 2
Enter the final states
q1
q2
Enter the number of alphabets: 2
Enter the alphabets
a
b
Enter the transitions
q0 a q1
q0 b q0
q1 a q2
q1 b q1
q2 a q1
q2 b q2
q3 a q1
q3 b q2

Minimised DFA

States:
{ q1  q2 }
{ q0 }

Transition table:
{ q1  q2 } a { q1  q2 }
{ q1  q2 } b { q1  q2 }
{ q0 } a { q1  q2 }
```

{ q0 } b { q0 }

Start state: { q0 }

Final states:
{ q1 q2 }

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name : Lex Analyser using Lex */

%{
#include <string.h>
int comment = 0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

\/\/.* {printf("\n%s is a COMMENT", yytext);}

\/\* {comment = 1; printf("\n"); ECHO; printf("\nCOMMENT BEGINS");}
\*\/ {comment = 0; printf("\nCOMMENT ENDS\n"); ECHO;}

#.* {printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}

int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {if(!comment) printf("\n%s is a KEYWORD", yytext);}

{identifier}\( {yytext[strlen(yytext) - 1] = '\0'; if(!comment) printf("\n%s is a FUNCTION\n\n( is
an OPENING PARANTHESIS", yytext);}

\{ {if(!comment) {printf("\n\n"); ECHO; printf("\nBLOCK BEGINS");}}

\} {if(!comment) {printf("\nBLOCK ENDS\n"); ECHO;}}
```

```
{identifier}(\[[0-9]*\])? {if(!comment) printf("\n%s is an IDENTIFIER", yytext);}

\".*\" {if(!comment) printf("\n%s is a STRING", yytext);}

[0-9]+ {if(!comment) printf("\n%s is a NUMBER", yytext);}

\( {if(!comment) printf("\n%s is an OPENING PARANTHESIS", yytext);}

\) {if(!comment) printf("\n%s is a CLOSING PARANTHESIS", yytext);}

\)(\:)? {if(!comment) printf("\n"); if(!comment) ECHO; if(!comment) printf("\n");}

= {if(!comment) printf("\n%s is an ASSIGNMENT OPERATOR", yytext);}

\<= |
\>= |
\< |
== |
\> {if(!comment) printf("\n%s is a RELATIONAL OPERATOR", yytext);}

, {if(!comment) printf("\n%s is a SEPARATOR", yytext);}

; {if(!comment) printf("\n%s is a DELIMITER", yytext);}

[+-/*%] {if(!comment) printf("\n%s is an ARITHMETIC OPERATOR", yytext);}

[|&\^] {if(!comment) printf("\n%s is a BITWISE OPERATOR", yytext);}

! |
&& |
\|\| {if(!comment) printf("\n%s is a LOGICAL OPERATOR", yytext);}

%%

int main(int argc, char **argv) {
        char filename[50];

        printf("Enter filename: ");
        scanf("%s", filename);

        FILE *file;
        file = fopen(filename,"r");

        if(!file) {
                printf("Could not open the file\n");
                exit(0);
```

```
        }

        yyin = file;
        yylex();
        printf("\n");
        return(0);
}

int yywrap() {
        return(1);
}
```

INPUT

in.c

```
//input program for lexical analyser

#include<stdio.h>
/* This
is
comment
*/
void main() {
        int a,b,c;
        a = 1;
        b = 2;
        c = a + b;
        printf("Sum: %d", c);
}
```

OUTPUT

```
Enter filename: in.c

//input program for lexical analyser is a COMMENT

#include<stdio.h> is a PREPROCESSOR DIRECTIVE

/*
COMMENT BEGINS



COMMENT ENDS
*/
```

void is a KEYWORD
main is a FUNCTION

(is an OPENING PARANTHESIS
) is a CLOSING PARANTHESIS

{
BLOCK BEGINS

int is a KEYWORD
a is an IDENTIFIER
, is a SEPARATOR
b is an IDENTIFIER
, is a SEPARATOR
c is an IDENTIFIER
; is a DELIMITER

a is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
1 is a NUMBER
; is a DELIMITER

b is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
2 is a NUMBER
; is a DELIMITER

c is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
a is an IDENTIFIER
+ is an ARITHMETIC OPERATOR
b is an IDENTIFIER
; is a DELIMITER

printf is a FUNCTION

(is an OPENING PARANTHESIS
"Sum: %d" is a STRING
, is a SEPARATOR
c is an IDENTIFIER
) is a CLOSING PARANTHESIS
; is a DELIMITER

BLOCK ENDS
}

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name :  Frequency of Lines, Words and Characters */

%{
int nlines, nwords, nchars;
%}

%%

\n { nchars++; nlines++; }

[^ \n\t]+ { nwords++, nchars = nchars + yyleng; }

. { nchars++; }

%%

int yywrap(void)
{
	return 1;
}

int main()
{
	char filename[50];
	printf("Enter filename: ");
	scanf("%s", filename);
	yyin = fopen(filename, "r");

	if(!yyin) {
		printf("Could not open file\n");
		exit(1);
	}

	yylex();
	printf("Lines = %d\nWords = %d\nChars = %d\n", nlines, nwords, nchars);

	return 0;
}
```

INPUT

in

Hi this is a sample

text file
for testing purposes
of programs using
Lex tool

OUTPUT

Enter filename: in
Lines = 5
Words = 15
Chars = 78

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name :  Valid Arithmetic expression */
```

lex

```
%{

   #include "y.tab.h"

%}

%%

[a-zA-Z_][a-zA-Z_0-9]* return id;

[0-9]+(\.[0-9]*)?      return num;

[+/*]                  return op;

.                     return yytext[0];

\n                     return 0;

%%

int yywrap()

{

return 1;

}
```

yacc

```
%{

   #include<stdio.h>

   int valid=1;

%}

%token num id op
```

```
%%
start : id '=' s ';'
s :     id x
      | num x
      | '-' num x
      | '(' s ')' x
      ;
x :     op s
      | '-' s
      |
      ;
%%
int yyerror()
{
   valid=0;
   printf("\nInvalid expression!\n");
   return 0;
}
int main()
{
   printf("Enter the expression:\n");
   yyparse();
   if(valid)
```

```
    {
        printf("\nValid expression.\n");
    }
}
```

OUTPUT

```
Enter the expression:
a=b+c;

Valid expression.

Enter the expression:
a=b+c-d;

Valid expression.

Enter the expression:
a=b

Invalid expression!

Enter the expression:
a=b@d;

Invalid expression!

Enter the expression:
a=b;

Valid expression.
```

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name :  Valid Identifier*/
```

lex

```
%{

   #include "y.tab.h"

%}

%%

[a-zA-Z_][a-zA-Z_0-9]* return letter;

[0-9]                    return digit;

.                    return yytext[0];

\n                    return 0;

%%

int yywrap()

{

return 1;

}
```

yacc

```
%{

   #include<stdio.h>

   int valid=1;

%}

%token digit letter

%%
```

```
start : letter s
s :     letter s
      | digit s
      |
      ;
%%
int yyerror()
{
   printf("\nInvalid identifier!\n");
   valid=0;
   return 0;
}
int main()
{
   printf("Enter identifier: ");
   yyparse();
   if(valid)
   {
      printf("\nValid identifier.\n");
   }
}
```

OUTPUT

```
Enter identifier: abc
```

Valid identifier.

Enter identifier: a

Valid identifier.

Enter identifier: 8ab

Invalid identifier!

Enter identifier: a`b

Invalid identifier!

Enter identifier: _

Valid identifier.

PROGRAM

/*Name : BHAGYA A JAI

Roll No : B21CSB18

Experiment Name : First and Follow */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct prod {
	char lhs[20];
	char rhs[20];
} prod;
typedef struct firstStruct {
	char arr[10];
	int count;
} firstStruct;
typedef struct followStruct {
	char arr[10];
	int count;
} followStruct;
int n;
prod grammar[50];
char temps[] = "PQRUVWXYZ";
int tempIndex = 0;

int contains(firstStruct* f, char c) {
	for(int i = 0; i < f->count; i++) {
		if(f->arr[i] == c)
			return 1;
	}

	return 0;
}
firstStruct* first(char c) {
	firstStruct* f = (firstStruct*) malloc(sizeof(firstStruct));
	f->count = 0;
	int flag;

	for(int i = 0; i < n; i++) {
		if(grammar[i].lhs[0] == c) {
			if((grammar[i].rhs[0] >= 'A' && grammar[i].rhs[0] <= 'Z') ||
grammar[i].rhs[0] == 'e') { // RHS starts with non-terminal
				for(int j = 0; j < strlen(grammar[i].rhs); j++) {
					if(j != 0 && (grammar[i].rhs[j] < 'A' || grammar[i].rhs[j] >'Z'))
{
						f->arr[f->count++] = grammar[i].rhs[j];
						break;
					}

					if(grammar[i].rhs[j] != 'e') {
```

```
                        firstStruct* f1 = first(grammar[i].rhs[j]);

                        if(contains(f1, 'e')) {
                                for(int k = 0; k < f1->count; k++) {
                                        if(f1->arr[k] == 'e')
                                                continue;

                                        flag = 0;

                                        for(int l = 0; l < f->count; l++) {
                                                if(f->arr[l] == f1->arr[k]) {
                                                        flag = 1;
                                                }
                                        }

                                        if(!flag)
                                                f->arr[f->count++] = f1->arr[k];
                                }

                                if(grammar[i].rhs[j + 1] == '\0') {
                                        f->arr[f->count++] = 'e';
                                        break;
                                }

                                continue;
                        } else {
                                for(int k = 0; k < f1->count; k++) {
                                        flag = 0;

                                        for(int l = 0; l < f->count; l++) {
                                                if(f->arr[l] == f1->arr[k]) {
                                                        flag = 1;
                                                }
                                        }

                                        if(!flag)
                                                f->arr[f->count++] = f1->arr[k];
                                }

                                break;
                        }
                } else {
                        if(grammar[i].rhs[j + 1] == '\0') {
                                f->arr[f->count++] = 'e';
                                break;
                        }
                }
        }
} else { // RHS starts with terminal
        flag = 0;
        for(int j = 0; j < f->count; j++) {
                if(f->arr[j] == grammar[i].rhs[0]) {
```

```
                                        flag = 1;
                                    }
                                }

                                if(!flag)
                                    f->arr[f->count++] = grammar[i].rhs[0];
                            }
                        }
                }

                return f;
}
followStruct* follow(char c) {
        followStruct* f = (followStruct*) malloc(sizeof(followStruct));
        int flag;

        if(grammar[0].lhs[0] == c) {
                f->arr[f->count++] = '$';
        }

        for(int i = 0; i < n; i++) {
                for(int j = 0; j < strlen(grammar[i].rhs); j++) {
                        flag = 0;

                        if(grammar[i].rhs[j] == c) {
                                if(grammar[i].rhs[j + 1] == '\0') {
                                        if(grammar[i].lhs[0] == c)
                                                break;

                                        followStruct* f2 = follow(grammar[i].lhs[0]);

                                        for(int k = 0; k < f2->count; k++) {
                                                flag = 0;

                                                for(int l = 0; l < f->count; l++) {
                                                        if(f->arr[l] == f2->arr[k]) {
                                                                flag = 1;
                                                        }
                                                }

                                                if(!flag) {
                                                        f->arr[f->count++] = f2->arr[k];
                                                }
                                        }
                                } else if(grammar[i].rhs[j + 1] >= 'A' && grammar[i].rhs[j + 1] <= 'Z')
{
                                        firstStruct* f1 = first(grammar[i].rhs[j + 1]);

                                        int m = 2;

                                        while(grammar[i].rhs[j + m] != '\0' && contains(f1, 'e')) {
                                                for(int k = 0; k < f1->count; k++) {
```

```
                        if(f1->arr[k] == 'e')
                                continue;

                        flag = 0;

                        for(int l = 0; l < f->count; l++) {
                                if(f->arr[l] == f1->arr[k]) {
                                        flag = 1;
                                }
                        }

                        if(!flag) {
                                f->arr[f->count++] = f1->arr[k];
                        }
                }

                flag = 0;

                if(grammar[i].rhs[j + m] >= 'A' && grammar[i].rhs[j +
m] <= 'Z') {
                        f1 = first(grammar[i].rhs[j + m]);
                        m++;
                } else {
                        f->arr[f->count++] = grammar[i].rhs[j + m];
                        flag = 1;
                        break;
                }
        }

        if(!flag) {
                for(int k = 0; k < f1->count; k++) {
                        if(f1->arr[k] == 'e')
                                continue;

                        flag = 0;

                        for(int l = 0; l < f->count; l++) {
                                if(f->arr[l] == f1->arr[k]) {
                                        flag = 1;
                                }
                        }

                        if(!flag) {
                                f->arr[f->count++] = f1->arr[k];
                        }
                }

                if(grammar[i].rhs[j + m] == '\0') {
                        if(grammar[i].lhs[0] == c)
                                break;

                        followStruct* f2 = follow(grammar[i].lhs[0]);
```

```
                                                for(int k = 0; k < f2->count; k++) {
                                                        flag = 0;

                                                        for(int l = 0; l < f->count; l++) {
                                                                if(f->arr[l] == f2->arr[k]) {
                                                                        flag = 1;
                                                                }
                                                        }

                                                        if(!flag) {
                                                                f->arr[f->count++] = f2->arr[k];
                                                        }
                                                }
                                        }
                                }
                        } else {
                                f->arr[f->count++] = grammar[i].rhs[j + 1];
                        }
                }
        }
    }

    return f;
}
void main()
 {
    int t = 0;
    char non_terminals[10];

    printf("Enter the number of productions: ");
    scanf("%d", &n);

    printf("\nEnter the productions (Eg: S = aBc):\n");
    for(int i = 0; i < n; i++) {
        scanf("%s = %s", grammar[i].lhs, grammar[i].rhs);
    }

    int flag;
    // Adding the non-terminals
    for(int i = 0; i < n; i++) {
        flag = 0;

        for(int j = 0; j < t; j++) {
            if(non_terminals[j] == grammar[i].lhs[0])
                flag = 1;
        }
        if(!flag) {
            non_terminals[t++] = grammar[i].lhs[0];
        }

        for(int j = 0; j < strlen(grammar[i].rhs); j++) {
```

```
                        if(grammar[i].rhs[j] >= 'A' && grammar[i].rhs[j] <= 'Z') {
                                flag = 0;

                                for(int k = 0; k < t; k++) {
                                        if(non_terminals[k] == grammar[i].rhs[j])
                                                flag = 1;
                                }

                                if(!flag) {
                                        non_terminals[t++] = grammar[i].rhs[j];
                                }
                                continue;
                        }
                }
        }
        for(int i = 0; i < t; i++) {
                flag = 0;

                int temp = n;

                // Eliminating indirect left recursion
                while(!flag) {
                        flag = 1;

                        for(int j = 0; j < temp; j++) {
                                if(grammar[j].lhs[0] == non_terminals[i]) {
                                        if(grammar[j].rhs[0] >= 'A' && grammar[j].rhs[0] <= 'Z' &&
grammar[j].rhs[0] > non_terminals[i]) {
                                                flag = 0;

                                                for(int k = 0; k < temp; k++) {
                                                        if(grammar[k].lhs[0] == grammar[j].rhs[0]) {
                                                                grammar[n].lhs[0] = non_terminals[i];
                                                                grammar[n].lhs[1] = '\0';
                                                                strcpy(grammar[n].rhs, grammar[k].rhs);
                                                                strcat(grammar[n].rhs,
&grammar[j].rhs[1]);

                                                                n++;
                                                        }
                                                }

                                                for(int k = j; k + 1 < n; k++) {
                                                        strcpy(grammar[k].lhs, grammar[k + 1].lhs);
                                                        strcpy(grammar[k].rhs, grammar[k + 1].rhs);
                                                }

                                                temp--;
                                                n--;
                                                j--;
                                        }
                                }
                        }
```

```
		}

		// Eliminating direct left recursion
		for(int l = 0; l < n; l++) {
			if(grammar[l].lhs[0] == non_terminals[i] && grammar[l].lhs[0] ==
grammar[l].rhs[0]) {
				for(int j = 0; ; j++) {
					if(grammar[l].rhs[j + 1] == '\0') {
						grammar[l].rhs[j] = temps[tempIndex];
						break;
					}

					grammar[l].rhs[j] = grammar[l].rhs[j + 1];
				}

				for(int j = 0; j < n; j++) {
					if(j == l)
						continue;

					if(grammar[j].lhs[0] == grammar[l].lhs[0]) {
						for(int k = 0; ; k++) {
							if(grammar[j].rhs[k] == '\0') {
								grammar[j].rhs[k] = temps[tempIndex];
								grammar[j].rhs[k + 1] = '\0';
								break;
							}
						}
					}
				}

				grammar[l].lhs[0] = temps[tempIndex];
				grammar[n].lhs[0] = temps[tempIndex];
				grammar[n].lhs[1] = '\0';
				grammar[n].rhs[0] = 'e';
				grammar[n].rhs[1] = '\0';
				n++;
				tempIndex++;
			}
		}
	}

	printf("\nAfter eliminating direct and indirect left recursions\n");
	for(int i = 0; i < n; i++) {
		printf("%s = %s\n", grammar[i].lhs, grammar[i].rhs);
	}

	// Adding the new non-terminals
	for(int i = 0; i < n; i++) {
		flag = 0;

		for(int j = 0; j < t; j++) {
			if(non_terminals[j] == grammar[i].lhs[0])
```

```
                        flag = 1;
                }

                if(!flag) {
                        non_terminals[t++] = grammar[i].lhs[0];
                }

                for(int j = 0; j < strlen(grammar[i].rhs); j++) {
                        if(grammar[i].rhs[j] >= 'A' && grammar[i].rhs[j] <= 'Z') {
                                flag = 0;

                                for(int k = 0; k < t; k++) {
                                        if(non_terminals[k] == grammar[i].rhs[j])
                                                flag = 1;
                                }

                                if(!flag) {
                                        non_terminals[t++] = grammar[i].rhs[j];
                                }

                                continue;
                        }
                }
        }

        firstStruct* f;
        followStruct* fl;

        // FIRST
        printf("\n");
        for(int i = 0; i < t; i++) {
                printf("FIRST(%c):", non_terminals[i]);

                f = first(non_terminals[i]);

                for(int j = 0; j < f->count; j++) {
                        printf(" %c", f->arr[j]);
                }

                printf("\n");
        }

        // FOLLOW
        printf("\n");
        for(int i = 0; i < t; i++) {
                printf("FOLLOW(%c):", non_terminals[i]);

                fl = follow(non_terminals[i]);

                for(int j = 0; j < fl->count; j++) {
                        printf(" %c", fl->arr[j]);
                }
```

```
                printf("\n");
        }
}
```

OUTPUT

Enter the number of productions: 6

Enter the productions (Eg: S = aBc):
E = E+T
E = T
T = T*F
T = F
F = (E)
F = i

After eliminating direct and indirect left recursions
P = +TP
Q = *FQ
T = FQ
F = (E)
F = i
E = T*FP
E = FP
P = e
Q = e

FIRST(E): (i
FIRST(T): (i
FIRST(F): (i
FIRST(P): + e
FIRST(Q): * e

FOLLOW(E):)
FOLLOW(T): + $) *
FOLLOW(F): * + $)
FOLLOW(P): $)
FOLLOW(Q): + $) *

PROGRAM

```
/*Name : BHAGYA A JAI
Roll No : B21CSB18
Experiment Name : Recursive Descent Parser */

#include <stdio.h>
#include <string.h>
int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];
int main() {
	puts("Enter the string");
	scanf("%s", string);
	cursor = string;
	puts("");
	puts("Input		Action");
	puts("--------------------------------");
	if (E() && *cursor == '\0') {
		puts("--------------------------------");
		puts("String is successfully parsed");
		return 0;
	}
	else {
		puts("--------------------------------");
		puts("Error in parsing String");
		return 1;
	}
}
int E() {
	printf("%-16s E -> T E'\n", cursor);
	if (T()) {
		if (Edash())
			return 1;
		else
			return 0;
```

```
	}
	else
		return 0;
}
int Edash() {
	if (*cursor == '+') {
		printf("%-16s E' -> + T E'\n", cursor);
		cursor++;
		if (T()) {
			if (Edash())
				return 1;
			else
				return 0;
		}
		else
			return 0;
	}
	else {
		printf("%-16s E' -> $\n", cursor);
		return 1;
	}
}
int T() {
	printf("%-16s T -> F T'\n", cursor);
	if (F()) {
		if (Tdash())
			return 1;
		else
			return 0;
	}
	else
		return 0;
}
int Tdash() {
	if (*cursor == '*') {
```

```
                printf("%-16s T' -> * F T'\n", cursor);
                cursor++;
                if (F()) {
                        if (Tdash())
                                return 1;
                        else
                                return 0;
                }
                else
                        return 0;
        }
        else {
                printf("%-16s T' -> $\n", cursor);
                return 1;
        }
}
int F() {
        if (*cursor == '(') {
                printf("%-16s F -> ( E )\n", cursor);
                cursor++;
                if (E()) {
                        if (*cursor == ')') {
                                cursor++;
                                return 1;
                        }
                        else
                                return 0;
                }
                else
                        return 0;
        }
        else if (*cursor == 'i') {
                printf("%-16s F -> i\n", cursor);
                cursor++;
                return 1;
```

```
        }
        else
                return 0;
}
```

OUTPUT

```
Enter the string
(i+i)*i+i*i

Input               Action
--------------------------------
(i+i)*i+i*i      E -> T E'
(i+i)*i+i*i      T -> F T'
(i+i)*i+i*i      F -> ( E )
i+i)*i+i*i       E -> T E'
i+i)*i+i*i       T -> F T'
i+i)*i+i*i       F -> i
+i)*i+i*i        T' -> $
+i)*i+i*i        E' -> + T E'
i)*i+i*i         T -> F T'
i)*i+i*i         F -> i
)*i+i*i          T' -> $
)*i+i*i          E' -> $
*i+i*i           T' -> * F T'
i+i*i            F -> i
+i*i             T' -> $
+i*i             E' -> + T E'
i*i              T -> F T'
i*i              F -> i
*i               T' -> * F T'
i                F -> i
                 T' -> $
                 E' -> $
--------------------------------
String is successfully parsed
```

PROGRAM

/*Name : BHAGYA A JAI

Roll No : B21CSB18

Experiment Name : Shift Reduce parser */

```
#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] == 'i')
		{
			printf("%si", 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] == 'E' && stk[z + 1] == '+' && stk[z + 2] == 'E')
		{
			printf("%sE+E", 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] == 'E' && stk[z + 1] == '-' && stk[z + 2] == 'E')
		{

			printf("%sE-E", 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] == '(' && stk[z + 1] == 'E' && stk[z + 2] == ')')
                {
                        printf("%s(E)", ac);
                        stk[z] = 'E';
                        stk[z + 1] = '\0';
                        stk[z + 2] = '\0';
                        printf("\n$%s\t%s$\t", stk, a);
                        i = i - 2;
                }
        }
        return ;
}
int main()
{
        // Grammar for reference
        printf("GRAMMAR is -\nE->E+E \nE->E-E \nE->(E) \nE->i\n");

        // Taking user input string
        printf("\nEnter the input string: ");
        scanf("%s", a);

        c = strlen(a);

        strcpy(act,"SHIFT");

        printf("\nstack \t input \t action");

        printf("\n$\t%s$\t", a);
        // Parsing the string
        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();

        // Final check to see if the string was accepted
        if(stk[0] == 'E' && stk[1] == '\0')
                printf("Accept\n");
        else
                printf("Reject\n");
}
```

OUTPUT

GRAMMAR is -

E->E+E
E->E-E
E->(E)
E->i

Enter the input string: (i+i)-(i-i)

stack	input	action
$	(i+i)-(i-i)$	SHIFT
$(	i+i)-(i-i)$	SHIFT
$(i	+i)-(i-i)$	REDUCE TO E -> i
$(E	+i)-(i-i)$	SHIFT
$(E+	i)-(i-i)$	SHIFT
$(E+i	)-(i-i)$	REDUCE TO E -> i
$(E+E	)-(i-i)$	REDUCE TO E -> E+E
$(E	)-(i-i)$	SHIFT
$(E)	-(i-i)$	REDUCE TO E -> (E)
$E	-(i-i)$	SHIFT
$E-	(i-i)$	SHIFT
$E-(	i-i)$	SHIFT
$E-(i	-i)$	REDUCE TO E -> i
$E-(E	-i)$	SHIFT

$E-(E-	i)$	SHIFT
$E-(E-i	)$	REDUCE TO E -> i
$E-(E-E	)$	REDUCE TO E -> E-E
$E-(E	)$	SHIFT
$E-(E)	$	REDUCE TO E -> (E)
$E-E	$	REDUCE TO E -> E-E
$E	$	Accept

PROGRAM

/*Name : BHAGYA A JAI

Roll No : B21CSB18

Experiment Name : Constant Propagation */

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

typedef struct statement {
	char lhs[2];
	char rhs[30];
} statement;

typedef struct map {
	char c;
	char val[8];
} map;

map m[100];
int k = 0;

int isConstant(char str[]) {
	for(int i = 0; i < strlen(str); i++) {
		if(str[i] == '-') {
			if(i != 0)
				return 0;
		} else if(str[i] < '0' || str[i] > '9')
			return 0;
	}

	return 1;
}

int mapContains(char c) {
	for(int i = 0; i < k; i++) {
		if(m[i].c == c)
			return 1;
	}

	return 0;
}

void mapAdd(char c, char* val) {
	m[k].c = c;
	strcpy(m[k].val, val);
	k++;
}
char* mapGet(char c) {
	for(int i = 0; i < k; i++) {
```

```
                if(m[i].c == c)
                        return m[i].val;
        }
}
void propogate(char rhs[]) {
        for(int i = 0; i < strlen(rhs); i++) {
                if(mapContains(rhs[i])) {
                        char str[8];
                        strcpy(str, mapGet(rhs[i]));

                        for(int j = strlen(rhs) + strlen(str) - 1; j > i; j--) {
                                rhs[j] = rhs[j - strlen(str) + 1];
                        }

                        for(int j = 0; j < strlen(str); j++) {
                                rhs[i + j] = str[j];
                        }
                }
        }
}

void main() {
        int n;
        printf("Enter the number of statements: ");
        scanf("%d", &n);
        statement statements[n];
        printf("Enter the statements:\n");
        for(int i = 0; i < n; i++) {
                scanf("%s = %s", statements[i].lhs, statements[i].rhs);

                if(isConstant(statements[i].rhs)) {
                        mapAdd(statements[i].lhs[0], statements[i].rhs);
                }
                propogate(statements[i].rhs);
        }
        printf("\nAfter Constant Propogation\n");
        for(int i = 0; i < n; i++) {
                printf("%s = %s\n", statements[i].lhs, statements[i].rhs);
        }
}
```

OUTPUT

```
Enter the number of statements: 3
Enter the statements:
a = 30
b = 20-a/2
c = b*(30/a+2)-a
After Constant Propogation
a = 30
b = 20-30/2
c = b*(30/30+2)-30
```

PROGRAM

/*Name : BHAGYA A JAI

Roll No : B21CSB18

Experiment Name : Intermediate code generation */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct statement {
	char lhs[2];
	char rhs[30];
} statement;
int k = 0;
char temps[] = "pqrstuvwxyz";
statement statements1[30];
int n1 = 0;
typedef struct stack {
	int TOP;
	int SIZE;
	char *arr;
} stack;
typedef struct expression {
	char *infix;
	char *postfix;
} expression;
void push(stack *s, char x) {
	if(s->TOP >= s->SIZE - 1) {
		printf("Overflow!\n");
		exit(0);
	} else {
		s->arr[++s->TOP] = x;
	}
}
char pop(stack *s) {
	if(s->TOP == -1) {
		printf("Underflow!\n");
		exit(0);
	} else {
		char x = s->arr[s->TOP];
		s->TOP--;
		return x;
	}
}
int ISP(char X) {
	if(X == '+' || X == '-')
		return 2;
	else if(X == '*' || X == '/')
		return 4;
	else if(X >= '0' && X <= '9' || X >= 'a' && X<= 'z' || X >= 'A' && X<= 'Z')
		return 8;
	else if(X == '(')
```

```
		return 0;
}
int ICP(char X) {
	if(X == '+' || X == '-')
		return 1;
	else if(X == '*' || X == '/')
		return 3;
	else if(X >= '0' && X <= '9' || X >= 'a' && X<= 'z' || X >= 'A' && X<= 'Z')
		return 7;
	else if(X == '(')
		return 9;
	else if(X == ')')
		return 0;
}
void infix_to_postfix(expression *exp) {
	int i = 0, j = 0;
	stack s;

	s.TOP = -1;
	s.SIZE = strlen(exp->infix);
	s.arr = (char*) malloc(s.SIZE * sizeof(char));

	push(&s, '(');

	while(s.TOP > -1) {
		char X = pop(&s);

		if(exp->infix[i] == '(') {
			push(&s, X);
			push(&s, exp->infix[i]);
		} else if(exp->infix[i] == ')') {
			while(X != '(')
			{
				exp->postfix[j] = X;
				X = pop(&s);
				j++;
			}
		} else if(exp->infix[i] >= 'a' && exp->infix[i] <= 'z' || exp->infix[i] >= 'A' &&
exp->infix[i] <= 'Z' || exp->infix[i] >= '0' && exp->infix[i] <= '9') {
			push(&s, X);
			exp->postfix[j] = exp->infix[i];
			j++;
		} else if(exp->infix[i] == '+' || exp->infix[i] == '-' || exp->infix[i] == '*' || exp->infix[i]
== '/' || exp->infix[i] == '(' || exp->infix[i] == ')') {
			if(ISP(X) >= ICP(exp->infix[i])) {
				while(ISP(X) >=  ICP(exp->infix[i])) {
					exp->postfix[j] = X;
					X = pop(&s);
					j++;
				}
				push(&s, X);
				push(&s, exp->infix[i]);
```

```
                } else {
                        push(&s, X);
                        push(&s, exp->infix[i]);
                }
        } else if(exp->infix[i] != ' ') {
                printf("Invalid statement!\n");
                exit(0);
        }
        i++;
    }
}
void icg(char c, char* str) {
    if(strlen(str) < 2) {
        statements1[n1].lhs[0] = c;
        statements1[n1].lhs[1] = '\0';
        statements1[n1].rhs[0] =  str[0];
        statements1[n1].rhs[1] = '\0';
        n1++;

        return;
    }

    int i = 0;
    char temp[2];

    while(i < strlen(str)) {
        if(str[i] == '+' || str[i] == '-' || str[i] == '*' || str[i] == '/') {
            statements1[n1].rhs[0] = str[i - 2];
            statements1[n1].rhs[1] = str[i];
            statements1[n1].rhs[2] = str[i - 1];
            statements1[n1].rhs[3] = '\0';
            statements1[n1].lhs[0] = temps[k];
            statements1[n1].lhs[1] = '\0';
            n1++;

            str[i - 2] = temps[k++];

            for(int j = i - 1; j < strlen(str) - 2; j++) {
                str[j] = str[j + 2];
            }

            str[strlen(str) - 2] = '\0';
            i = -1;
        }

        i++;
    }
    statements1[n1].lhs[0] = c;
    statements1[n1].lhs[1] = '\0';

    statements1[n1].rhs[0] =  temps[k - 1];
    statements1[n1].rhs[1] = '\0';
```

```
        n1++;
}

void main() {
        int n;

        printf("Enter the number of statements: ");
        scanf("%d", &n);
        statement statements[n];

        printf("Enter the statements:\n");
        for(int i = 0; i < n; i++) {
                scanf("%s = %s", statements[i].lhs, statements[i].rhs);
        }

        expression exp;
        for(int i = 0; i < n; i++) {
                exp.infix = statements[i].rhs;
                exp.postfix = (char*) malloc(strlen(exp.infix) * sizeof(char));

                exp.infix[strlen(exp.infix)] = ')';

                infix_to_postfix(&exp);

                icg(statements[i].lhs[0], exp.postfix);
        }

        printf("\nIntermediate Code:\n");
        for(int i = 0; i < n1; i++) {
                printf("%s = %s\n", statements1[i].lhs, statements1[i].rhs);
        }
}
```

OUTPUT

```
Enter the number of statements: 3
Enter the statements:
a = b+c-c*d
b = a+b*(c-d)
c = d

Intermediate Code:
p = b+c
q = c*d
r = p-q
a = r
s = c-d
t = b*s
u = a+t
b = u
c = d
```

PROGRAM

```
/*Name : BHAGYA A JAI

Roll No : B21CSB18

Experiment Name : Assembly code generation */

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

typedef struct statement {
	char lhs[2];
	char rhs[4];
} statement;

void move(char a, char b) {
	printf("MOV AX, %c\n", b);
	printf("MOV %c, AX\n", a);
}

void add(char a, char b, char c) {
	printf("MOV AX, %c\n", b);
	printf("MOV BX, %c\n", c);
	printf("ADD AX, BX\n");
	printf("MOV %c, AX\n", a);
}

void sub(char a, char b, char c) {
	printf("MOV AX, %c\n", b);
	printf("MOV BX, %c\n", c);
	printf("SUB AX, BX\n");
	printf("MOV %c, AX\n", a);
}

void mul(char a, char b, char c) {
	printf("MOV AX, %c\n", b);
	printf("MOV BX, %c\n", c);
	printf("MUL AX, BX\n");
	printf("MOV %c, AX\n", a);
}

void div(char a, char b, char c) {
	printf("MOV AX, %c\n", b);
	printf("MOV BX, %c\n", c);
	printf("DIV AX, BX\n");
	printf("MOV %c, AX\n", a);
}

void main() {
	int n;

	printf("Enter the number of statements in Intermediate Code: ");
	scanf("%d", &n);
```

```
	statement statements[n];

	printf("Enter the statements:\n");
	for(int i = 0; i < n; i++) {
		scanf("%s = %s", statements[i].lhs, statements[i].rhs);
	}

	printf("\nGenerated Code:\n");
	for(int i = 0; i < n; i++) {
		if(strlen(statements[i].rhs) == 1) {
			move(statements[i].lhs[0], statements[i].rhs[0]);
		} else {
			switch(statements[i].rhs[1]) {
				case '+':
					add(statements[i].lhs[0], statements[i].rhs[0],
statements[i].rhs[2]);
					break;
				case '-':
					sub(statements[i].lhs[0], statements[i].rhs[0],
statements[i].rhs[2]);
					break;
				case '*':
					mul(statements[i].lhs[0], statements[i].rhs[0],
statements[i].rhs[2]);
					break;
				case '/':
					div(statements[i].lhs[0], statements[i].rhs[0],
statements[i].rhs[2]);
					break;
				default:
					printf("Invalid statement!\n");
					return;
			}
		}
	}
}
```

OUTPUT

```
Enter the number of statements in Intermediate Code: 9
Enter the statements:
p = b+c
q = c*d
r = p-q
a = r
s = c-d
t = b*s
u = a+t
b = u
c = d

Generated Code:
```

```
MOV AX, b
MOV BX, c
ADD AX, BX
MOV p, AX
MOV AX, c
MOV BX, d
MUL AX, BX
MOV q, AX
MOV AX, p
MOV BX, q
SUB AX, BX
MOV r, AX
MOV AX, r
MOV a, AX
MOV AX, c
MOV BX, d
SUB AX, BX
MOV s, AX
MOV AX, b
MOV BX, s
MUL AX, BX
MOV t, AX
MOV AX, a
MOV BX, t
ADD AX, BX
MOV u, AX
MOV AX, u
MOV b, AX
MOV AX, d
MOV c, AX
```