

Q1: Write a c-program to implement lexical analyser

Program:

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

// Function to check if a character is a delimiter
bool isDelimiter(char ch) {
    return (ch == '' || ch == '+' || ch == '-' || ch == '*' ||
           ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
           ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
           ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
           ch == '\n' || ch == 't');
}

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
           ch == '!' || ch == '<' || ch == '=');
}

// Function to check if a string is a keyword
bool isKeyword(char* str) {
    char keywords[32][10] = {"auto", "break", "case", "char", "const", "continue",
                           "default", "do", "double", "else", "enum", "extern",
                           "float", "for", "goto", "if", "int", "long", "register",
                           "return", "short", "signed", "sizeof", "static", "struct",
                           "switch", "typedef", "union", "unsigned", "void", "volatile",
                           "while"};
    int i;
    for (i = 0; i < 32; ++i) {
        if (strcmp(keywords[i], str) == 0) {
            return true;
        }
    }
    return false;
}

// Function to check if a string is a valid identifier
bool isValidIdentifier(char* str) {
    if (str == NULL || !isalpha(str[0])) {
        return false;
    }
    int i;
    for (i = 1; i < strlen(str); i++) {
        if (!isalnum(str[i])) {
```

```

        return false;
    }
}
return true;
}

// Function to perform lexical analysis
void lexicalAnalyzer(char* str) {
    int left = 0, right = 0;
    int len = strlen(str);
    char subStr[100]; // Assuming max token length 99

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false) {
            right++;
        }

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true) {
                printf("Operator: %c\n", str[right]);
            } else if (str[right] != ' ' && str[right] != '\n' && str[right] != '\t') {
                printf("Delimiter: %c\n", str[right]);
            }
            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right ||
                   (right == len && left != right)) {
            strncpy(subStr, str + left, right - left);
            subStr[right - left] = '\0';

            if (isKeyword(subStr) == true) {
                printf("Keyword: %s\n", subStr);
            } else if (isValidIdentifier(subStr) == true) {
                printf("Identifier: %s\n", subStr);
            } else if (isdigit(subStr[0]) && isValidIdentifier(subStr) == false) {
                printf("Literal (Numeric): %s\n", subStr);
            } else {
                printf("Invalid Token: %s\n", subStr);
            }
            left = right;
        }
    }
}

int main() {
    char input_string[1000];
    printf("Enter the C code to analyze:\n");
    fgets(input_string, sizeof(input_string), stdin);

    printf("\nTokens found:\n");
    lexicalAnalyzer(input_string);

    return 0;
}

```

Output:

Enter the C code to analyze:

int s=9;

Tokens found:

Keyword: int

Identifier: s

Operator: =

Literal (Numeric): 9

Delimiter: ;

Q2: Write a c-program to implement lexical analyser for generating keywords.

Program:

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

// List of keywords
char keywords[32][10] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "int", "long", "register", "return", "short", "signed", "sizeof",
    "static", "struct", "switch", "typedef", "union", "unsigned", "void",
    "volatile", "while"
};

// Function to check if a word is a keyword
int isKeyword(char *word) {
    for (int i = 0; i < 32; i++) {
        if (strcmp(keywords[i], word) == 0)
            return 1;
    }
    return 0;
}

int main() {
    char input[1000], word[50];
    int i = 0, j = 0;

    printf("Enter the source code:\n");
    fgets(input, sizeof(input), stdin);

    printf("\nKeywords found:\n");

    while (input[i] != '\0') {
        if (isalpha(input[i])) { // Start of a word
            j = 0;
            while (isalnum(input[i])) {
                word[j++] = input[i++];
            }
            word[j] = '\0';

            if (isKeyword(word))
                printf("%s\n", word);
        } else {
            i++;
        }
    }

    return 0;
}
```

Output:

Enter the source code:

```
int main() {  
    float x, y;  
    if (x < y)  
        return y;  
}
```

Keywords found:

```
int  
float  
if  
return
```

Q3: Write a c-program to implement first and follow functions.

Program:

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

#define MAX 20

int n; // Number of productions
char production[MAX][MAX];
char firstSet[MAX][MAX];
char followSet[MAX][MAX];
char nonTerminals[MAX];
int m = 0; // Number of non-terminals

void findFirst(char, int, int);
void findFollow(char);
int isTerminal(char);
int isNonTerminal(char);
void addToResult(char[], char);

int main() {
    int i, j;
    char c, ch;

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

    printf("Enter the productions (e.g., E=TR):\n");
    for (i = 0; i < n; i++) {
        scanf("%s", production[i]);
        nonTerminals[m++] = production[i][0];
    }

    printf("\n--- FIRST sets ---\n");
    for (i = 0; i < n; i++) {
        findFirst(production[i][0], 0, 0);
        printf("FIRST(%c) = { %s }\n", production[i][0], firstSet[production[i][0] - 'A']);
    }

    printf("\n--- FOLLOW sets ---\n");
    for (i = 0; i < n; i++) {
        findFollow(production[i][0]);
        printf("FOLLOW(%c) = { %s }\n", production[i][0], followSet[production[i][0] - 'A']);
    }

    return 0;
}

// Function to find FIRST set of a given non-terminal
void findFirst(char c, int q1, int q2) {
    int j;
    char subResult[20];
    subResult[0] = '\0';
```

```

if (!isupper(c)) {
    addToResult(firstSet[c - 'A'], c);
    return;
}

for (j = 0; j < n; j++) {
    if (production[j][0] == c) {
        if (production[j][2] == '#')
            addToResult(firstSet[c - 'A'], '#');
        else if (islower(production[j][2]))
            addToResult(firstSet[c - 'A'], production[j][2]);
        else
            findFirst(production[j][2], q1, q2);
    }
}
}

// Function to find FOLLOW set of a non-terminal
void findFollow(char c) {
    int i, j, k;

    // Add '$' to FOLLOW(start symbol)
    if (production[0][0] == c)
        addToResult(followSet[c - 'A'], '$');

    for (i = 0; i < n; i++) {
        for (j = 2; j < strlen(production[i]); j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    if (isupper(production[i][j + 1])) {
                        strcat(followSet[c - 'A'], firstSet[production[i][j + 1] - 'A']);
                    } else {
                        addToResult(followSet[c - 'A'], production[i][j + 1]);
                    }
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                    findFollow(production[i][0]);
            }
        }
    }
}

// Helper function to add symbol to result set
void addToResult(char result[], char val) {
    int i;
    for (i = 0; result[i] != '\0'; i++) {
        if (result[i] == val)

```

```
        return;  
    }  
    result[i] = val;  
    result[i + 1] = '\0';  
}
```

Output:

Enter number of productions: 5
Enter the productions (e.g., E=TR):

E =TE`

E` =+TE`| e

T=FT`

T` =*FT`|e

F=(E)|id

--- FIRST sets ---

FIRST(E) = { (, id}

FIRST(E`) = { (, id}

FIRST(T) = { (, id}

FIRST(T`) = { *, e }

FIRST(F) = { (, id}

--- FOLLOW sets ---

FOLLOW(E) = { \$,) }

FOLLOW(E`) = { \$,) }

FOLLOW(T) = { +,\$,) }

FOLLOW(F) = { +,\$) }

FOLLOW(T`) = { +,\$), * }

Q4: Write a c-program to implement LL(1) / predictive parser.

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 10

// Grammar:
// E→TE'
// E'→+TE'|ε
// T→FT'
// T'→*FT'|ε
// F→(E)|id

// Terminals: +, *, (, ), id, $
char stack[50];
int top = -1;

void push(char c) {
    stack[++top] = c;
}

char pop() {
    if(top == -1)
        return '\0';
    return stack[top--];
}

char peek() {
    if(top == -1)
        return '\0';
    return stack[top];
}

// Function to display the parsing step
void displayStackAndInput(char *input, int index) {
    printf("\nStack: ");
    for (int i = 0; i <= top; i++) printf("%c", stack[i]);
    printf("\nInput: %s\t", &input[index]);
}

int main() {
    char input[50];
    int i = 0;

    printf("Grammar used:\n");
```

```

printf("E -> T E\n");
printf("E' -> + T E' | ε\n");
printf("T -> F T\n");
printf("T' -> * F T' | ε\n");
printf("F -> (E) | i\n\n");

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

push('$');
push('E');

printf("\nParsing Steps:\n");

while (input[i] != '\0') {
    displayStackAndInput(input, i);
    char topSymbol = peek();
    char currentInput = input[i];

    if (topSymbol == currentInput) {
        printf("Action: Match %c", currentInput);
        pop();
        i++;
    }
    else if (topSymbol == 'E') {
        printf("Action: E → T E");
        pop();
        push(' ');
        push('T');
    }
    else if (topSymbol == ' ') { // Represents E'
        if (currentInput == '+') {
            printf("Action: E' → + T E");
            pop();
            push(' ');
            push('T');
            push('+');
        } else {
            printf("Action: E' → ε");
            pop();
        }
    }
    else if (topSymbol == 'T') {
        printf("Action: T → F T");
        pop();
        push(' ');
        push('F');
    }
    else if (topSymbol == '') { // Represents T'
        if (currentInput == '*') {
            printf("Action: T' → * F T");
            pop();
            push(' ');
            push('F');
            push('*');
        }
    }
}

```

```
    }Else{
        printf("Action: T' → ε");
        pop();
    }
}
else if (topSymbol == 'F') {
    if (currentInput == 'i') {
        printf("Action: F → i");
        pop();
        push('i');
    } else if (currentInput == '(') {
        printf("Action: F → (E)");
        pop();
        push(')');
        push('E');
        push('(');
    } else {
        printf("Error: Invalid symbol for F");
        exit(0);
    }
}
else {
    printf("Error: Invalid symbol or mismatch");
    exit(0);
}

if (top == -1 && input[i] == '$') {
    printf("\n\nString accepted successfully!\n");
    return 0;
}
}

if (top == -1 && input[i] == '\0')
    printf("\n\nString accepted successfully!\n");
else
    printf("\n\nString rejected!\n");

return 0;
}
```

OUTPUT:

Grammar used:

$$\begin{aligned} E &\rightarrow T E' \\ E &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow * F T' \mid \epsilon \\ i & \end{aligned}$$

Enter the input string (end with \$): i*i+i\$ Parsing

Steps:

Stack: \$E	Input: i*i+i\$	Action: $E \rightarrow T E'$
Stack: '\$T	Input: i*i+i\$	Action: $T \rightarrow F T'$
Stack: \$"F	Input: i*i+i\$	Action: $F \rightarrow i$
Stack: \$"i	Input: i*i+i\$	Action: Match i
Stack: \$"	Input: *i+i\$	Action: $T' \rightarrow * F T'$
Stack: \$"F*	Input: *i+i\$	Action: Match *
Stack: \$"F	Input: i+i\$	Action: $F \rightarrow i$
Stack: \$"i	Input: i+i\$	Action: Match i
Stack: \$"	Input: +i\$	Action: $T' \rightarrow \epsilon$
Stack: \$'	Input: +i\$	Action: $E' \rightarrow + T E'$
Stack: '\$T+	Input: +i\$	Action: Match +
Stack: '\$T	Input: i\$	Action: $T \rightarrow F T'$
Stack: \$"F	Input: i\$	Action: $F \rightarrow i$
Stack: \$"i	Input: i\$	Action: Match i
Stack: \$"	Input: \$	Action: $T' \rightarrow \epsilon$
Stack: \$'	Input: \$	Action: $E' \rightarrow \epsilon$
Stack: \$	Input: \$	Action: Match \$

String accepted successfully!

Q5: Write a c-program to implement SLR Parser.

Program:

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

#define MAX 50

// Grammar used:
// 1. E→E+T
// 2. E→T
// 3. T→T*F
// 4. T→F
// 5. F→(E)
// 6. F→i
//
// Input symbols: i + * ( ) $
// Start symbol: E

// Productions for reductions
char *production[] = {
    " ", // index starts at 1
    "E>E+T", // 1
    "E>T", // 2
    "T>T*F", // 3
    "T>F", // 4
    "F>(E)", // 5
    "F>i" // 6
};

// Parser stack
int stack[MAX];
int top = 0;

// Parsing table lookup function
const char *parsingTable[12][9] = {
    {"S5", "", "", "S4", "", "", "1", "2", "3"}, // S5 to S4, S4 to 1, 2, 3
    {"", "S6", "", "", "", "Acc", "", "", ""}, // S6 to Acc
    {"", "R2", "S7", "", "R2", "R2", "", "", ""}, // R2 to S7, S7 to R2
    {"", "R4", "R4", "", "R4", "R4", "", "", ""}, // R4 to R4
    {"S5", "", "", "S4", "", "", "8", "2", "3"}, // S5 to S4, S4 to 8, 2, 3
    {"", "R6", "R6", "", "R6", "R6", "", "", ""}, // R6 to R6
    {"S5", "", "", "S4", "", "", "9", "3"}, // S5 to S4, S4 to 9, 3
    {"S5", "", "", "S4", "", "", "", "", "10"}, // S5 to S4, S4 to 10
    {"", "S6", "", "", "S11", "", "", "", ""}, // S6 to S11
    {"", "R1", "S7", "", "R1", "R1", "", "", ""} // R1 to S7, S7 to R1
};
```

```

{ "", "R3", "R3", "", "R3", "R3", "", "", "" },
{ "", "R5", "R5", "", "R5", "R5", "", "", "" }
};

// Function to get column index for terminal/nonterminal
int getSymbolIndex(char symbol) {
    switch (symbol) {
        case 'i': return 0;
        case '+': return 1;
        case '*': return 2;
        case '(': return 3;
        case ')': return 4;
        case '$': return 5;
        case 'E': return 6;
        case 'T': return 7;
        case 'F': return 8;
        default: return -1;
    }
}

int main() {
    char input[50];
    char symbol;
    int i = 0;

    printf("Grammar:\n");
    for (int k = 1; k <= 6; k++)
        printf("%d. %s\n", k, production[k]);

    printf("\nEnter input string (end with $): ");
    scanf("%s", input);

    stack[top] = 0; // Initial state

    printf("\nStack\tInput\tAction\n");
    printf("-----\n");

    while (1) {
        int state = stack[top];
        symbol = input[i];
        int col = getSymbolIndex(symbol);

        if (col == -1) {
            printf("Error: Invalid symbol %c\n", symbol);
            exit(1);
        }

        const char *action = parsingTable[state][col];

        // Print current stack and input
        for (int k = 0; k <= top; k++) printf("%d ", stack[k]);
        printf("\t%s\t", &input[i]);

        if (strcmp(action, "") == 0) {
            printf("Error: No action found!\n");
        }
    }
}

```

```
exit(1);
} else if (action[0] == 'S') { // SHIFT
    printf("Shift %c, go to state %c\n", symbol, action[1]);
    top++;
    stack[top] = symbol;
    top++;
    stack[top] = action[1] - '0';
    i++;
} else if (action[0] == 'R') { // REDUCE
    int prodNo = action[1] - '0';
    char *rhs = strchr(production[prodNo], '>') + 1;
    int len = strlen(rhs);

    // Pop stack for each symbol on RHS
    for (int k = 0; k < len * 2; k++)
        top--;

    char lhs = production[prodNo][0];
    state = stack[top];
    int lhsCol = getSymbolIndex(lhs);
    const char *goTo = parsingTable[state][lhsCol];

    printf("Reduce by %s, Go to state %s\n", production[prodNo], goTo);

    top++;
    stack[top] = lhs;
    top++;
    stack[top] = atoi(goTo);
} else if (strcmp(action, "Acc") == 0) {
    printf("Accept!\nInput string accepted successfully.\n");
    break;
} else {
    printf("Error: Invalid action!\n");
    exit(1);
}
}

return 0;
}
```

Output:

Grammar:

1. E->E+T
2. E->T
3. T->T*F
4. T->F
5. F->(E)
6. F->i

Enter input string (end with \$): i+i*i\$

Stack	Input	Action
0	i+i*i\$	Shift i, go to state 5
0 i 5	+i*i\$	Reduce by F->i, Go to state 3
0 F 3	+i*i\$	Reduce by T->F, Go to state 2
0 T 2	+i*i\$	Reduce by E->T, Go to state 1
0 E 1	+i*i\$	Shift +, go to state 6
0 E 1 + 6	i*i\$	Shift i, go to state 5
0 E 1 + 6 i 5	*i\$	Reduce by F->i, Go to state 10
0 E 1 + 6 F 10	*i\$	Reduce by T->T*F, Go to state 9
0 E 1 + 6 T 9	\$	Reduce by E->E+T, Go to state 1
0 E 1	\$	Accept!

Input string accepted successfully

Q6: Write a lex program to count the number of vowels and consonants in the string.

Program:

```
%{  
int vowels = 0, consonants = 0;  
%}  
  
%%%  
[aeiouAEIOU]      { vowels++;  
} [a-zA-Z]  
consonants++; }  
.\\n      /* ignore other characters */  
%%%  
  
int main() {  
    printf("Enter a string: ");  
    yylex();  
    printf("\\nNumber of vowels: %d", vowels);  
    printf("\\nNumber of consonants: %d\\n", consonants);  
    return 0;  
}  
  
int yywrap() {  
    return 1;  
}
```

Output:

```
Enter a string:  
Hello World  
Number of vowels: 3  
Number of consonants: 7
```

Q7: Write a lex program for line count ,word count and character count.

Program:

```
%{  
int lnno=0,wordno=0,charno=0;  
%}  
word [.*.*\t]  
eol [\n]  
%%  
{word} {wordno++; charno+=yyleng;}  
{eol} {charno++;lnno++;wordno++;}  
. {charno++;}  
%%  
main()  
{  
printf("Enter the String:");  
yylex();  
printf("Line number= %d\n",lnno);  
printf("Word number= %d\n",wordno);  
printf("Character number= %d\n",charno);  
return 0;  
}
```

Output:

```
Enter the String:  
Hello World  
Line number= 13  
Word number= 43  
Character number= 232
```

Q8: Write a lex program to print “digit” whenever it receives the token from 0-9.

Program:

```
%{  
#include<stdio.h>;  
%}  
%%  
[0-9] {printf("Digit ");}  
%%  
int yywrap(void){}  
void main()  
{  
printf("Enter the String:");  
yylex();  
}
```

Output:

Enter the String:

```
8  
Digit
```

Q9: Write a lex program print * whenever it receives the token “3cse1”.

Program:

```
%{  
#include<stdio.h>;  
%}  
%%  
"3csec" {printf("*");}  
%%  
int yywrap(void){}  
void main()  
{  
printf("Enter the String:");  
yylex();  
}
```

Output:

Enter the String:

```
3csec3csec3csec  
***
```

Q10: Write a lex program for keyword recognition.

Program:

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

int keyword_count = 0;

// List of C keywords
char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto", "if",
    "int", "long", "register", "return", "short", "signed", "sizeof",
    "static", "struct", "switch", "typedef", "union", "unsigned", "void",
    "volatile", "while"
};

int isKeyword(char *word) {
    for (int i = 0; i < 32; i++) {
        if (strcmp(keywords[i], word) == 0)
            return 1;
    }
    return 0;
}
%}

%%%
[a-zA-Z_][a-zA-Z0-9_]* {
    if (isKeyword(yytext)) {
        printf("Keyword: %s\n", yytext);
        keyword_count++;
    }
}
.\n      /* ignore other characters */

%%

int main() {
    printf("Enter the C code (Ctrl+D to end input):\n\n");
    yylex();
    printf("\nTotal number of keywords: %d\n", keyword_count);
    return 0;
}

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

Output:

Enter the C code (Ctrl+D to end input):

```
int main() {  
    int a, b;  
    float c;  
    if(a > b)  
        return 1;  
}
```

Keyword: int

Keyword: int

Keyword: float

Keyword: if

Keyword: return

Total number of keywords: 5