

**PRACTICAL SESSION PLAN**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.NO** | **DAY** | **ANALYTICAL**  **(8.00 AM TO 11.AM)** | **PRACTICAL**  **(12.00 PM TO 2.00 PM)** | **CLASS TEST**  **(2.00 PM TO 3.00PM)** |
| **1** | **DAY1** | **UNIT-1**  **-** **Phases of a compiler**  **-** **Tokens**  **-** **Lexical errors**  **-** **Regular Expression**  **-** **Regular Language**  **Complete the Lab Installation of Lex** | 1. **Write a LEX program to identify the capital words from the given input.**   **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **%%**  **[A-Z]+ { printf("Capital word: %s\n", yytext); }**  **.|\n { /\* ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv)**  **{**  **yylex();**  **return 0;**  **}**  **C-PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **int main() {**  **char input[100];**  **printf("Enter a string: ");**  **fgets(input, sizeof(input), stdin);**  **printf("Capital letters: ");**  **for(int i = 0; input[i] != '\0'; i++) {**  **if(isupper(input[i])) {**  **putchar(input[i]);**  **}**  **}**  **printf("\n");**  **return 0;**  **}**   1. **Write a LEX program to check whether the given input is digit or not.**   **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **%%**  **[0-9] { printf("%s is a digit\n", yytext); }**  **. { printf("%s is not a digit\n", yytext); }**  **%%**  **int main() {**  **yylex();**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **int main() {**  **char input;**  **printf("Enter a character: ");**  **scanf("%c", &input);**  **if (isdigit(input)) {**  **printf("The character '%c' is a digit.\n", input);**  **} else {**  **printf("The character '%c' is not a digit.\n", input);**  **}**  **return 0;**  **}**   1. **The Company ABC runs with employees with several departments. The Organization manager had all the mobile numbers of employees. Assume that you are the manager and need to verify the valid mobile numbers because there may be some invalid numbers present. Implement a LEX program to check whether the mobile number is valid or not.**   **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <ctype.h>**  **%}**  **%%**  **[0-9]{10} { printf("Valid mobile number: %s\n", yytext); }**  **.|\n { printf("Invalid mobile number: %s\n", yytext); }**  **%%**  **int main() {**  **yylex();**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <ctype.h>**  **int isValidMobileNumber(const char\* number) {**  **// Check if the length is exactly 10**  **if (strlen(number) != 10) {**  **return 0;**  **}**    **// Check if all characters are digits**  **for (int i = 0; i < 10; i++) {**  **if (!isdigit(number[i])) {**  **return 0;**  **}**  **}**  **return 1;**  **}**  **int main() {**  **char mobileNumber[20];**  **printf("Enter the mobile number: ");**  **scanf("%s", mobileNumber);**  **if (isValidMobileNumber(mobileNumber)) {**  **printf("The mobile number is valid.\n");**  **} else {**  **printf("The mobile number is not valid.\n");**  **}**  **return 0;**  **}**   1. **In a class, an English teacher was teaching the vowels and consonants to the students. She says “Vowel sounds allow the air to flow freely, causing the chin to drop noticeably, whilst consonant sounds are produced by restricting the air flow”. As a class activity the students are asked to identify the vowels and consonants in the given word/sentence and count the number of elements in each. Write an algorithm to help the student to count the number of vowels and consonants in the given sentence.**   **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **%%**  **[aeiouAEIOU] { printf("%c is a vowel\n", yytext[0]); }**  **[a-zA-Z] { printf("%c is a consonant\n", yytext[0]); }**  **. { /\* Ignore other characters \*/ }**  **%%**  **int main() {**  **printf("Enter a string: ");**  **yylex();**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **void identifyVowelsAndConsonants(char str[]);**  **int main() {**  **char str[100];**  **printf("Enter a string: ");**  **fgets(str, sizeof(str), stdin);**  **identifyVowelsAndConsonants(str);**  **return 0;**  **}**  **void identifyVowelsAndConsonants(char str[]) {**  **int i = 0;**  **while (str[i] != '\0') {**  **char ch = tolower(str[i]);**  **if (isalpha(ch)) {**  **if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {**  **printf("%c is a vowel\n", str[i]);**  **} else {**  **printf("%c is a consonant\n", str[i]);**  **}**  **}**  **i++;**  **}**  **}**   1. **Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. In general there are 32 keywords. The prime function of Lexical Analyser is token Generation. Among the 6 types of tokens, differentiating Keyword and Identifier is a challenging issue. Thus write a LEX program to separate keywords and identifiers.**   **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **/\* List of C keywords \*/**  **char \*keywords[] = {**  **"auto", "double", "int", "struct", "break", "else", "long", "switch",**  **"case", "enum", "register", "typedef", "char", "extern", "return",**  **"union", "const", "float", "short", "unsigned", "continue", "for",**  **"signed", "void", "default", "goto", "sizeof", "volatile", "do",**  **"if", "static", "while"**  **};**  **/\* Function to check if a given string is a keyword \*/**  **int is\_keyword(char \*str) {**  **for(int i = 0; i < 32; i++) {**  **if(strcmp(str, keywords[i]) == 0) {**  **return 1;**  **}**  **}**  **return 0;**  **}**  **%}**  **%%**  **/\* Rule to match keywords \*/**  **[a-zA-Z\_][a-zA-Z0-9\_]\* {**  **if(is\_keyword(yytext)) {**  **printf("Keyword: %s\n", yytext);**  **} else {**  **printf("Identifier: %s\n", yytext);**  **}**  **}**  **/\* Rule to match whitespace and ignore it \*/**  **[ \t\n]+ { /\* Ignore whitespace \*/ }**  **/\* Rule to match any other character \*/**  **. { /\* Ignore any other character \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <ctype.h>**  **// Array of C keywords**  **const 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"**  **};**  **const int numKeywords = sizeof(keywords) / sizeof(keywords[0]);**  **// Function to check if a string is a keyword**  **int isKeyword(const char \*str) {**  **for (int i = 0; i < numKeywords; i++) {**  **if (strcmp(str, keywords[i]) == 0) {**  **return 1;**  **}**  **}**  **return 0;**  **}**  **// Function to check if a string is a valid identifier**  **int isIdentifier(const char \*str) {**  **if (!isalpha(str[0]) && str[0] != '\_') {**  **return 0;**  **}**  **for (int i = 1; str[i] != '\0'; i++) {**  **if (!isalnum(str[i]) && str[i] != '\_') {**  **return 0;**  **}**  **}**  **return 1;**  **}**  **// Function to tokenize the input string and classify tokens**  **void classifyTokens(const char \*input) {**  **char token[100];**  **int i = 0, j = 0;**  **while (input[i] != '\0') {**  **if (isalnum(input[i]) || input[i] == '\_') {**  **token[j++] = input[i];**  **} else {**  **if (j != 0) {**  **token[j] = '\0';**  **if (isKeyword(token)) {**  **printf("Keyword: %s\n", token);**  **} else if (isIdentifier(token)) {**  **printf("Identifier: %s\n", token);**  **} else {**  **printf("Unknown: %s\n", token);**  **}**  **j = 0;**  **}**  **}**  **i++;**  **}**  **// Check for the last token**  **if (j != 0) {**  **token[j] = '\0';**  **if (isKeyword(token)) {**  **printf("Keyword: %s\n", token);**  **} else if (isIdentifier(token)) {**  **printf("Identifier: %s\n", token);**  **} else {**  **printf("Unknown: %s\n", token);**  **}**  **}**  **}**  **int main() {**  **char input[1000];**  **printf("Enter the code:\n");**  **fgets(input, sizeof(input), stdin);**  **classifyTokens(input);**  **return 0;**  **}**  **6.Write a LEX program to identify and count positive and negative numbers.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **int positive\_count = 0;**  **int negative\_count = 0;**  **%}**  **%%**  **[0-9]+ { positive\_count++; }**  **-[0-9]+ { negative\_count++; }**  **\n { /\* Ignore newline characters \*/ }**  **. { /\* Ignore all other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **yylex(); // Start scanning the input**  **printf("Number of positive numbers: %d\n", positive\_count);**  **printf("Number of negative numbers: %d\n", negative\_count);**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **int main() {**  **int number;**  **int positiveCount = 0, negativeCount = 0;**  **printf("Enter numbers (0 to stop):\n");**  **while (1) {**  **printf("Enter a number: ");**  **scanf("%d", &number);**  **// Check for the sentinel value to stop the input**  **if (number == 0) {**  **break;**  **}**    **// Count positive and negative numbers**  **if (number > 0) {**  **positiveCount++;**  **} else if (number < 0) {**  **negativeCount++;**  **}**  **}**  **// Display the results**  **printf("Number of positive numbers: %d\n", positiveCount);**  **printf("Number of negative numbers: %d\n", negativeCount);**  **return 0;**  **}**  **7.Write a LEX program to recognise numbers and words in a statement. Pooja is a small girl of age 3 always fond of games. Due to the pandemic, she was not allowed to play outside. So her mother designs a gaming event by showing a flash card. Pooja has to separate the numbers in one list and words in another list shown in the flash card.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **/\* Define patterns for numbers and words \*/**  **NUMBER [0-9]+**  **WORD [a-zA-Z]+**  **%%**  **{NUMBER} { printf("NUMBER: %s\n", yytext); }**  **{WORD} { printf("WORD: %s\n", yytext); }**  **%%**  **int main() {**  **yylex(); // Call the lexical analyzer**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <string.h>**  **#define MAX\_LENGTH 1000**  **void separateNumbersAndWords(const char \*statement, char \*numbers, char \*words) {**  **int numIndex = 0, wordIndex = 0;**  **int inNumber = 0, inWord = 0;**  **for (int i = 0; i < strlen(statement); i++) {**  **char ch = statement[i];**  **if (isdigit(ch)) {**  **if (!inNumber) {**  **if (inWord) {**  **words[wordIndex++] = ' ';**  **inWord = 0;**  **}**  **inNumber = 1;**  **}**  **numbers[numIndex++] = ch;**  **} else if (isalpha(ch)) {**  **if (!inWord) {**  **if (inNumber) {**  **numbers[numIndex++] = ' ';**  **inNumber = 0;**  **}**  **inWord = 1;**  **}**  **words[wordIndex++] = ch;**  **} else {**  **if (inNumber) {**  **numbers[numIndex++] = ' ';**  **inNumber = 0;**  **}**  **if (inWord) {**  **words[wordIndex++] = ' ';**  **inWord = 0;**  **}**  **}**  **}**    **if (inNumber) numbers[numIndex++] = '\0'; // End of number list**  **else numbers[numIndex] = '\0'; // Null-terminate the string**  **if (inWord) words[wordIndex++] = '\0'; // End of word list**  **else words[wordIndex] = '\0'; // Null-terminate the string**  **}**  **int main() {**  **char statement[MAX\_LENGTH];**  **char numbers[MAX\_LENGTH] = {0};**  **char words[MAX\_LENGTH] = {0};**  **printf("Enter a statement: ");**  **fgets(statement, sizeof(statement), stdin);**    **// Remove newline character if present**  **size\_t len = strlen(statement);**  **if (len > 0 && statement[len - 1] == '\n') {**  **statement[len - 1] = '\0';**  **}**  **separateNumbersAndWords(statement, numbers, words);**  **printf("Numbers: %s\n", numbers);**  **printf("Words: %s\n", words);**  **return 0;**  **}**  **8.Write a LEX program to accept string starting with vowel.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **%%**  **^[aeiouAEIOU][a-zA-Z]\* { printf("Accepted: %s\n", yytext); }**  **. { /\* Ignore non-matching input \*/ }**  **%%**  **int main() {**  **yylex(); // Call the lexer**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h> // For tolower function**  **#define MAX\_LENGTH 1000**  **int main() {**  **char str[MAX\_LENGTH];**    **printf("Enter a string: ");**  **fgets(str, sizeof(str), stdin);**    **// Remove newline character if present**  **size\_t len = strlen(str);**  **if (len > 0 && str[len - 1] == '\n') {**  **str[len - 1] = '\0';**  **}**  **// Check if the first character is a vowel**  **char firstChar = tolower(str[0]); // Convert to lowercase to handle both cases**  **if (firstChar == 'a' || firstChar == 'e' || firstChar == 'i' ||**  **firstChar == 'o' || firstChar == 'u') {**  **printf("The string starts with a vowel: %s\n", str);**  **} else {**  **printf("The string does not start with a vowel.\n");**  **}**  **return 0;**  **}** | **Unit -1** |
| **2** | **DAY 2** | **UNIT – II**  **-CFG**  **-CFL**  **-Derivation**  **-Parse tree**  **-Ambiguity**  **-Shift Reduce Parser**  **-Operator Precedence Parser**  **-SLR Parser**  **-Back tracking**  **-Recursive Descent Parser**  **-Predictive parser**  **-LL Parser** | **9.Write a LEX program to find the length of the longest word.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **#define MAX\_WORD\_LENGTH 1000**  **int maxLength = 0;**  **int currentLength = 0;**  **char longestWord[MAX\_WORD\_LENGTH + 1];**  **%}**  **%%**  **[a-zA-Z]+ {**  **currentLength = yyleng;**  **if (currentLength > maxLength) {**  **maxLength = currentLength;**  **strncpy(longestWord, yytext, currentLength);**  **longestWord[currentLength] = '\0';**  **}**  **}**  **\n|\r\n|\r { /\* Ignore newlines and carriage returns \*/ }**  **. { /\* Ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **yylex();**  **printf("Longest word length: %d\n", maxLength);**  **printf("Longest word: %s\n", longestWord);**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <string.h>**  **#define MAX\_LENGTH 1000**  **int main() {**  **char sentence[MAX\_LENGTH];**  **char longestWord[MAX\_LENGTH];**  **int maxLength = 0;**  **int currentLength = 0;**  **char currentWord[MAX\_LENGTH];**    **printf("Enter a sentence: ");**  **fgets(sentence, sizeof(sentence), stdin);**    **// Remove newline character if present**  **size\_t len = strlen(sentence);**  **if (len > 0 && sentence[len - 1] == '\n') {**  **sentence[len - 1] = '\0';**  **}**    **// Iterate through each character in the sentence**  **for (int i = 0; i <= len; i++) {**  **if (isalpha(sentence[i])) {**  **currentWord[currentLength++] = sentence[i];**  **} else {**  **// End of the current word**  **if (currentLength > 0) {**  **currentWord[currentLength] = '\0';**  **if (currentLength > maxLength) {**  **maxLength = currentLength;**  **strcpy(longestWord, currentWord);**  **}**  **currentLength = 0;**  **}**  **}**  **}**  **// Check the last word if there was no trailing space**  **if (currentLength > 0) {**  **currentWord[currentLength] = '\0';**  **if (currentLength > maxLength) {**  **maxLength = currentLength;**  **strcpy(longestWord, currentWord);**  **}**  **}**  **printf("The longest word is: %s\n", longestWord);**  **printf("Length of the longest word: %d\n", maxLength);**  **return 0;**  **}**  **10.** **A networking company wants to validate the URL for their clients. Write a LEX program to implement the same.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **void yyerror(const char \*s) {**  **fprintf(stderr, "Error: %s\n", s);**  **}**  **%}**  **%%**  **^(http|https)://**  **{**  **BEGIN(HTTP\_URL);**  **printf("URL Scheme: %s\n", yytext);**  **}**  **<HTTP\_URL>^[a-zA-Z0-9\-\.]+**  **{**  **printf("Domain Name: %s\n", yytext);**  **}**  **<HTTP\_URL>^[a-zA-Z0-9\-\.]+**  **{**  **printf("Domain: %s\n", yytext);**  **}**  **<HTTP\_URL>^[/:]**  **{**  **printf("Path Separator: %s\n", yytext);**  **}**  **<HTTP\_URL>^[a-zA-Z0-9\-.\_~:/?#[\]@!$&'()\*+,;=]+**  **{**  **printf("Path or Query: %s\n", yytext);**  **}**  **<HTTP\_URL>^[ \t\r\n]+**  **{**  **// Ignore whitespace**  **}**  **<HTTP\_URL>. // Handle any other character**  **{**  **printf("Invalid URL character: %s\n", yytext);**  **yyerror("Invalid character in URL.");**  **}**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("Error opening file");**  **exit(1);**  **}**  **yyin = file;**  **}**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <ctype.h>**  **#define MAX\_URL\_LENGTH 2048**  **// Function to check if the URL starts with a valid scheme (http:// or https://)**  **int hasValidScheme(const char \*url) {**  **return (strstr(url, "http://") == url || strstr(url, "https://") == url);**  **}**  **// Function to check if the URL contains a valid domain**  **int hasValidDomain(const char \*url) {**  **const char \*domain = url;**    **// Skip past the scheme**  **if (strstr(url, "http://") == url) {**  **domain += 7;**  **} else if (strstr(url, "https://") == url) {**  **domain += 8;**  **}**    **// Check if there's a domain after the scheme**  **if (\*domain == '\0' || \*domain == '/') {**  **return 0;**  **}**  **// Check if domain contains at least one dot (.)**  **if (strchr(domain, '.') == NULL) {**  **return 0;**  **}**  **return 1;**  **}**  **// Function to validate the URL**  **int isValidURL(const char \*url) {**  **if (strlen(url) == 0 || strlen(url) > MAX\_URL\_LENGTH) {**  **return 0;**  **}**  **if (!hasValidScheme(url) || !hasValidDomain(url)) {**  **return 0;**  **}**  **return 1;**  **}**  **int main() {**  **char url[MAX\_URL\_LENGTH];**  **printf("Enter a URL: ");**  **fgets(url, sizeof(url), stdin);**  **// Remove newline character if present**  **size\_t len = strlen(url);**  **if (len > 0 && url[len - 1] == '\n') {**  **url[len - 1] = '\0';**  **}**  **if (isValidURL(url)) {**  **printf("The URL is valid.\n");**  **} else {**  **printf("The URL is not valid.\n");**  **}**  **return 0;**  **}**  **11.School management wants to validate DOB of all students. Write a LEX program to implement it.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **// Helper function to check if a number is within a given range**  **int is\_in\_range(int num, int min, int max) {**  **return (num >= min && num <= max);**  **}**  **// Helper function to check if a date is valid**  **int is\_valid\_date(int day, int month, int year) {**  **// Days per month (not accounting for leap years)**  **int days\_in\_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};**    **// Check for leap year and adjust February days**  **if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {**  **days\_in\_month[1] = 29;**  **}**  **// Validate month and day**  **if (is\_in\_range(month, 1, 12) && is\_in\_range(day, 1, days\_in\_month[month - 1])) {**  **return 1;**  **}**  **return 0;**  **}**  **%}**  **%%**  **[0-3][0-9]/[0-1][0-9]/[0-9][0-9][0-9][0-9] {**  **int day = atoi(yytext);**  **int month = atoi(yytext + 3);**  **int year = atoi(yytext + 6);**  **if (is\_valid\_date(day, month, year)) {**  **printf("Valid DOB: %s\n", yytext);**  **} else {**  **printf("Invalid DOB: %s\n", yytext);**  **}**  **}**  **.|\n {**  **// Ignore other characters and newlines**  **}**  **%%**  **int main() {**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <stdbool.h>**  **#include <stdlib.h>**  **// Function to check if a year is a leap year**  **bool isLeapYear(int year) {**  **if (year % 4 == 0) {**  **if (year % 100 == 0) {**  **if (year % 400 == 0) {**  **return true;**  **}**  **return false;**  **}**  **return true;**  **}**  **return false;**  **}**  **// Function to check if a given date is valid**  **bool isValidDate(int day, int month, int year) {**  **// Check year range**  **if (year < 1900 || year > 2100) {**  **return false;**  **}**  **// Check month range**  **if (month < 1 || month > 12) {**  **return false;**  **}**  **// Days in each month**  **int daysInMonth[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };**  **// Adjust for leap year**  **if (isLeapYear(year)) {**  **daysInMonth[1] = 29;**  **}**  **// Check day range**  **if (day < 1 || day > daysInMonth[month - 1]) {**  **return false;**  **}**  **return true;**  **}**  **int main() {**  **char dateStr[11];**  **int day, month, year;**  **printf("Enter date of birth (dd/mm/yyyy): ");**  **fgets(dateStr, sizeof(dateStr), stdin);**  **// Parse the input date string**  **if (sscanf(dateStr, "%d/%d/%d", &day, &month, &year) != 3) {**  **printf("Invalid format. Please enter date in dd/mm/yyyy format.\n");**  **return 1;**  **}**  **// Validate the date**  **if (isValidDate(day, month, year)) {**  **printf("The date of birth is valid.\n");**  **} else {**  **printf("The date of birth is not valid.\n");**  **}**  **return 0;**  **}**  **12.** **Write a LEX program to recognize a word and relational operator.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <ctype.h>**  **%}**  **%%**  **[ \t\n]+ ; // Ignore whitespace (spaces, tabs, newlines)**  **if|else|while|for|return { printf("Keyword: %s\n", yytext); }**  **==|!=|<|>|<=|>= { printf("Relational operator: %s\n", yytext); }**  **[a-zA-Z\_][a-zA-Z\_0-9]\* { printf("Word: %s\n", yytext); }**  **. { /\* Ignore other characters \*/ }**  **%%**  **int main() {**  **yylex();**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <ctype.h>**  **#define MAX\_LENGTH 1000**  **// Function to check if a string is a relational operator**  **int isRelationalOperator(const char \*str) {**  **return (strcmp(str, "==") == 0 || strcmp(str, "!=") == 0 ||**  **strcmp(str, "<") == 0 || strcmp(str, ">") == 0 ||**  **strcmp(str, "<=") == 0 || strcmp(str, ">=") == 0);**  **}**  **// Function to process input and recognize words and relational operators**  **void processInput(const char \*input) {**  **char word[MAX\_LENGTH];**  **int wordIndex = 0;**  **printf("Words:\n");**  **printf("Relational Operators:\n");**  **for (int i = 0; i <= strlen(input); i++) {**  **char ch = input[i];**    **if (isalnum(ch) || ch == '\_') { // Building a word**  **word[wordIndex++] = ch;**  **} else {**  **if (wordIndex > 0) {**  **word[wordIndex] = '\0';**  **if (isRelationalOperator(word)) {**  **printf("%s (Relational Operator)\n", word);**  **} else {**  **printf("%s (Word)\n", word);**  **}**  **wordIndex = 0;**  **}**  **// Handle relational operators separately**  **if (ch == '=' && input[i + 1] == '=') {**  **printf("== (Relational Operator)\n");**  **i++; // Skip the next '='**  **} else if (ch == '!' && input[i + 1] == '=') {**  **printf("!= (Relational Operator)\n");**  **i++; // Skip the next '='**  **} else if (ch == '<' || ch == '>' || ch == '=' || ch == '!') {**  **printf("%c (Relational Operator)\n", ch);**  **}**  **}**  **}**  **}**  **int main() {**  **char input[MAX\_LENGTH];**  **printf("Enter a statement: ");**  **fgets(input, sizeof(input), stdin);**  **// Remove newline character if present**  **size\_t len = strlen(input);**  **if (len > 0 && input[len - 1] == '\n') {**  **input[len - 1] = '\0';**  **}**  **processInput(input);**  **return 0;**  **}**  **13.** **Write a LEX code to replace a word with another word in a file.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **#define WORD\_TO\_REPLACE "foo"**  **#define REPLACEMENT\_WORD "bar"**  **%}**  **%%**  **{WORD\_TO\_REPLACE} { printf("%s", REPLACEMENT\_WORD); }**  **.|\n { putchar(yytext[0]); }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("fopen");**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <string.h>**  **#define MAX\_LINE\_LENGTH 1024**  **// Function to replace occurrences of a word in a string**  **void replaceWordInString(char \*str, const char \*oldWord, const char \*newWord) {**  **char buffer[MAX\_LINE\_LENGTH];**  **char \*pos;**  **int oldWordLen = strlen(oldWord);**  **int newWordLen = strlen(newWord);**  **int bufferLen = 0;**  **// Continue searching for oldWord in str**  **while ((pos = strstr(str, oldWord)) != NULL) {**  **// Copy the part before oldWord**  **bufferLen = pos - str;**  **strncpy(buffer, str, bufferLen);**  **buffer[bufferLen] = '\0';**  **// Append the newWord to buffer**  **strcat(buffer, newWord);**  **// Append the part after oldWord**  **strcat(buffer, pos + oldWordLen);**  **// Copy buffer back to str**  **strcpy(str, buffer);**  **}**  **}**  **// Function to replace word in a file**  **void replaceWordInFile(const char \*filename, const char \*oldWord, const char \*newWord) {**  **FILE \*file = fopen(filename, "r+");**  **if (file == NULL) {**  **perror("Error opening file");**  **exit(EXIT\_FAILURE);**  **}**  **char line[MAX\_LINE\_LENGTH];**  **fseek(file, 0, SEEK\_SET); // Move to the beginning of the file**  **long pos;**  **while ((pos = ftell(file)) >= 0 && fgets(line, sizeof(line), file)) {**  **// Replace oldWord with newWord in the current line**  **replaceWordInString(line, oldWord, newWord);**  **// Move to the beginning of the line to overwrite it**  **fseek(file, pos, SEEK\_SET);**  **fprintf(file, "%s", line);**  **// Move to the next line**  **fseek(file, pos + strlen(line), SEEK\_SET);**  **}**  **fclose(file);**  **}**  **int main() {**  **char filename[256];**  **char oldWord[256];**  **char newWord[256];**  **printf("Enter the filename: ");**  **scanf("%255s", filename);**  **printf("Enter the word to replace: ");**  **scanf("%255s", oldWord);**  **printf("Enter the new word: ");**  **scanf("%255s", newWord);**  **replaceWordInFile(filename, oldWord, newWord);**  **printf("Replacement complete.\n");**  **return 0;**  **}**  **14.** **A School student was asked to do basic mathematical operations. Implement a LEX program to implement the same.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **int yylex();**  **void yyerror(const char \*s);**  **%}**  **%option noyywrap**  **NUM [0-9]+**  **OP [\+\-\\*/]**  **WS [ \t\n]+**  **%%**  **{NUM} {**  **yylval = atoi(yytext); // Convert string to integer**  **return NUMBER;**  **}**  **{OP} {**  **return yytext[0]; // Return the operator**  **}**  **{WS} {**  **// Ignore whitespace**  **}**  **. {**  **yyerror("Invalid character");**  **}**  **%%**  **int main() {**  **printf("Enter an expression (e.g., 3 + 5):\n");**  **yyparse(); // Call parser (if using yacc/bison)**  **return 0;**  **}**  **void yyerror(const char \*s) {**  **fprintf(stderr, "Error: %s\n", s);**  **}**  **C PRO:**  **#include <stdio.h>**  **// Function prototypes**  **void add(double a, double b);**  **void subtract(double a, double b);**  **void multiply(double a, double b);**  **void divide(double a, double b);**  **int main() {**  **double num1, num2;**  **int choice;**  **printf("Enter first number: ");**  **scanf("%lf", &num1);**  **printf("Enter second number: ");**  **scanf("%lf", &num2);**  **printf("Select operation:\n");**  **printf("1. Addition\n");**  **printf("2. Subtraction\n");**  **printf("3. Multiplication\n");**  **printf("4. Division\n");**  **printf("Enter your choice (1/2/3/4): ");**  **scanf("%d", &choice);**  **switch (choice) {**  **case 1:**  **add(num1, num2);**  **break;**  **case 2:**  **subtract(num1, num2);**  **break;**  **case 3:**  **multiply(num1, num2);**  **break;**  **case 4:**  **divide(num1, num2);**  **break;**  **default:**  **printf("Invalid choice.\n");**  **break;**  **}**  **return 0;**  **}**  **// Function to perform addition**  **void add(double a, double b) {**  **printf("Result: %.2lf\n", a + b);**  **}**  **// Function to perform subtraction**  **void subtract(double a, double b) {**  **printf("Result: %.2lf\n", a - b);**  **}**  **// Function to perform multiplication**  **void multiply(double a, double b) {**  **printf("Result: %.2lf\n", a \* b);**  **}**  **// Function to perform division**  **void divide(double a, double b) {**  **if (b != 0) {**  **printf("Result: %.2lf\n", a / b);**  **} else {**  **printf("Error: Division by zero is not allowed.\n");**  **}**  **}**  **15.** **Write a LEX Program to check the email address is valid or not.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **%}**  **%%**  **^[a-zA-Z0-9.\_]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ { printf("Valid email address.\n"); }**  **.|\n { /\* Ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <stdbool.h>**  **// Function to check if the email address is valid**  **bool isValidEmail(const char \*email) {**  **int atCount = 0;**  **bool hasDot = false;**    **// Check if the email is empty**  **if (strlen(email) == 0) {**  **return false;**  **}**  **// Check the email structure**  **for (int i = 0; i < strlen(email); i++) {**  **if (email[i] == '@') {**  **atCount++;**  **} else if (email[i] == '.') {**  **hasDot = true;**  **}**  **}**    **// Valid email must contain exactly one '@' and at least one '.'**  **if (atCount == 1 && hasDot) {**  **return true;**  **}**  **return false;**  **}**  **int main() {**  **char email[256];**  **printf("Enter email address: ");**  **scanf("%255s", email);**  **if (isValidEmail(email)) {**  **printf("The email address is valid.\n");**  **} else {**  **printf("The email address is not valid.\n");**  **}**  **return 0;**  **}**  **16.** **Write a LEX Program to convert the substring abc to ABC from the given input string.**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **%}**  **%%**  **abc { printf("ABC"); }**  **. { putchar(yytext[0]); }**  **%%**  **int main(int argc, char \*\*argv) {**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#define MAX\_LENGTH 1000**  **// Function to replace occurrences of a substring with another substring**  **void replaceSubstring(char \*str, const char \*oldSub, const char \*newSub) {**  **char buffer[MAX\_LENGTH];**  **char \*pos;**  **int oldSubLen = strlen(oldSub);**  **int newSubLen = strlen(newSub);**  **int bufferLen = 0;**  **while ((pos = strstr(str, oldSub)) != NULL) {**  **// Copy the part before oldSub**  **bufferLen = pos - str;**  **strncpy(buffer, str, bufferLen);**  **buffer[bufferLen] = '\0';**  **// Append the newSub to buffer**  **strcat(buffer, newSub);**  **// Append the part after oldSub**  **strcat(buffer, pos + oldSubLen);**  **// Copy buffer back to str**  **strcpy(str, buffer);**  **}**  **}**  **int main() {**  **char input[MAX\_LENGTH];**  **printf("Enter a string: ");**  **fgets(input, sizeof(input), stdin);**  **// Remove newline character if present**  **size\_t len = strlen(input);**  **if (len > 0 && input[len - 1] == '\n') {**  **input[len - 1] = '\0';**  **}**  **// Replace all occurrences of "abc" with "ABC"**  **replaceSubstring(input, "abc", "ABC");**  **printf("Modified string: %s\n", input);**  **return 0;**  **}** | **Unit -2** |
| **3** | **DAY 3** | **UNIT – III**  **-Synthesized Attribute**  **-Inherited Attribute**  **-Construction of Syntax tree**  **-Bottom up Evaluation of S & L-Attributed Definition**  **-Top Down Translation** | **17.The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Write a LEX specification file to take input C program from a .c file and count t he number of characters, number of lines & number of words.**  **Input Source Program: (sample.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **int char\_count = 0;**  **int line\_count = 0;**  **int word\_count = 0;**  **int in\_word = 0;**  **void count\_characters(const char \*yytext) {**  **char\_count += strlen(yytext);**  **}**  **void count\_words() {**  **if (in\_word == 0) {**  **in\_word = 1;**  **word\_count++;**  **}**  **}**  **%}**  **%%**  **[ \t\n]+ { /\* Ignore spaces, tabs, and new lines \*/ }**  **"/\*"[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*/ { /\* Ignore multi-line comments \*/ }**  **"//".\* { /\* Ignore single-line comments \*/ }**  **[a-zA-Z\_][a-zA-Z0-9\_]\* { count\_words(); count\_characters(yytext); }**  **. { count\_characters(yytext); }**  **\n { line\_count++; in\_word = 0; }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **fprintf(stderr, "Cannot open file %s\n", argv[1]);**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **if (argc > 1) {**  **fclose(yyin);**  **}**  **printf("Number of characters: %d\n", char\_count);**  **printf("Number of lines: %d\n", line\_count);**  **printf("Number of words: %d\n", word\_count);**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#include <stdio.h>**  **int main()**  **{**  **int number1, number2, sum;**  **printf("Enter two integers: ");**  **scanf("%d %d", &number1, &number2);**  **sum = number1 + number2;**  **printf("%d + %d = %d", number1, number2, sum);**  **return 0;**  **}**  **18.** **Write a LEX program to print all the constants in the given C source program file.**  **Input Source Program: (sample.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <stdlib.h>**  **%}**  **%%**  **[ \t\n]+ { /\* Ignore whitespace \*/ }**  **"/\*"[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*/ { /\* Ignore multi-line comments \*/ }**  **"//".\* { /\* Ignore single-line comments \*/ }**  **0[xX][0-9a-fA-F]+ { printf("Hexadecimal constant: %s\n", yytext); }**  **[0-9]+\.[0-9]\*([eE][-+]?[0-9]+)? { printf("Floating-point constant: %s\n", yytext); }**  **\.[0-9]+([eE][-+]?[0-9]+)? { printf("Floating-point constant: %s\n", yytext); }**  **[0-9]+[eE][-+]?[0-9]+ { printf("Floating-point constant: %s\n", yytext); }**  **[0-9]+ { printf("Integer constant: %s\n", yytext); }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **fprintf(stderr, "Cannot open file %s\n", argv[1]);**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **if (argc > 1) {**  **fclose(yyin);**  **}**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#define PI 3.14**  **#include<stdio.h> #include<conio.h>**  **void main()**  **{**  **int a,b,c = 30;**  **printf("hello");**  **}**  **19.Write a LEX program to count the number of Macros defined and header files included in the C program.**  **Input Source Program: (sample.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **int macro\_count = 0;**  **int header\_count = 0;**  **%}**  **%%**  **"/\*"[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*/ { /\* Ignore multi-line comments \*/ }**  **"//".\* { /\* Ignore single-line comments \*/ }**  **"#define" { macro\_count++; }**  **"#include" { header\_count++; }**  **.|\n { /\* Ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **fprintf(stderr, "Cannot open file %s\n", argv[1]);**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **if (argc > 1) {**  **fclose(yyin);**  **}**  **printf("Number of macros defined: %d\n", macro\_count);**  **printf("Number of header files included: %d\n", header\_count);**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **C PRO:**  **#define PI 3.14**  **#include<stdio.h>**  **#include<conio.h>**  **void main()**  **{**  **int a,b,c = 30;**  **printf("hello");**  **}**  **20.** **Write a LEX program to print all HTML tags in the input file.**  **Input Source Program: (sample.html)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **%}**  **%%**  **<!--(.|\n)\*?--> { /\* Ignore HTML comments \*/ }**  **<[^>]+> { printf("HTML tag: %s\n", yytext); }**  **.|\n { /\* Ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **fprintf(stderr, "Cannot open file %s\n", argv[1]);**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **if (argc > 1) {**  **fclose(yyin);**  **}**  **return 0;**  **}**  **int yywrap() {**  **return 1;**  **}**  **HTML PRO:**  **<html>**  **<body>**  **<h1>My First Heading</h1>**  **<p>My first paragraph.</p>**  **</body>**  **</html>**  **21.** **Write a LEX program which adds line numbers to the given C program file and display the same in the standard output.**  **Input Source Program: (sample.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **int line\_num = 1;**  **%}**  **%%**  **\n { printf("%d: \n", line\_num++); }**  **.|\t { ECHO; }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("Failed to open file");**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#define PI 3.14**  **#include<stdio.h>**  **#include<conio.h>**  **void main()**  **{**  **int a,b,c = 30;**  **printf("hello");**  **}**  **22.** **Write a LEX program to count the number of comment lines in a given C program and eliminate them and write into another file.**  **Input Source File: (input.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **int comment\_lines = 0;**  **FILE \*outfile;**  **%}**  **%%**  **"//".\*\n { comment\_lines++; /\* Single-line comment \*/ }**  **"/\*"[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*"/" {**  **char \*p = yytext;**  **while ((p = strstr(p, "\n")) != NULL) {**  **comment\_lines++;**  **p++;**  **}**  **}**  **.|\n { fputc(yytext[0], outfile); }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc != 3) {**  **fprintf(stderr, "Usage: %s <input\_file> <output\_file>\n", argv[0]);**  **return 1;**  **}**  **FILE \*infile = fopen(argv[1], "r");**  **if (!infile) {**  **perror("Failed to open input file");**  **return 1;**  **}**  **outfile = fopen(argv[2], "w");**  **if (!outfile) {**  **perror("Failed to open output file");**  **fclose(infile);**  **return 1;**  **}**  **yyin = infile;**  **yylex();**  **fclose(infile);**  **fclose(outfile);**  **printf("Number of comment lines: %d\n", comment\_lines);**  **return 0;**  **}**  **C PRO:**  **#include<stdio.h>**  **int main()**  **{**  **int a,b,c; /\*varible declaration\*/**  **printf(“enter two numbers”);**  **scanf(“%d %d”,&a,&b);**  **c=a+b;//adding two numbers**  **printf(“sum is %d”,c);**  **return 0;**  **}**  **23.** **Implement Lexical Analyzer using LEX or FLEX (Fast Lexical Analyzer). The program should separate the tokens in the given C program and display with appropriate caption.**  **Input Source Program: (sample.c)**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **// Define token types**  **enum {**  **KEYWORD,**  **IDENTIFIER,**  **CONSTANT,**  **STRING\_LITERAL,**  **PUNCTUATOR,**  **OPERATOR**  **};**  **void print\_token(const char\* token, int type);**  **%}**  **// Define keywords**  **%{**  **const 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", NULL**  **};**  **int is\_keyword(const char\* str) {**  **for (int i = 0; keywords[i] != NULL; i++) {**  **if (strcmp(str, keywords[i]) == 0) {**  **return 1;**  **}**  **}**  **return 0;**  **}**  **%}**  **%%**  **[ \t\n]+ /\* ignore whitespace \*/**  **"//".\* /\* ignore single-line comments \*/**  **"/\*"[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*"/" /\* ignore multi-line comments \*/**  **\"([^\\\n]|(\\.))\*?\" { print\_token(yytext, STRING\_LITERAL); }**  **0[xX][0-9a-fA-F]+ { print\_token(yytext, CONSTANT); }**  **[0-9]+("."[0-9]\*)?([eE][+-]?[0-9]+)? { print\_token(yytext, CONSTANT); }**  **[A-Za-z\_][A-Za-z0-9\_]\* {**  **if (is\_keyword(yytext)) {**  **print\_token(yytext, KEYWORD);**  **} else {**  **print\_token(yytext, IDENTIFIER);**  **}**  **}**  **[~!%^&\*()\-+=|\\{}[\]:;\"'<>,.?/] { print\_token(yytext, PUNCTUATOR); }**  **. { print\_token(yytext, OPERATOR); }**  **%%**  **void print\_token(const char\* token, int type) {**  **switch (type) {**  **case KEYWORD:**  **printf("Keyword: %s\n", token);**  **break;**  **case IDENTIFIER:**  **printf("Identifier: %s\n", token);**  **break;**  **case CONSTANT:**  **printf("Constant: %s\n", token);**  **break;**  **case STRING\_LITERAL:**  **printf("String Literal: %s\n", token);**  **break;**  **case PUNCTUATOR:**  **printf("Punctuator: %s\n", token);**  **break;**  **case OPERATOR:**  **printf("Operator: %s\n", token);**  **break;**  **default:**  **printf("Unknown: %s\n", token);**  **break;**  **}**  **}**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("Failed to open file");**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **return 0;**  **}**  **C PRO:**  **#include<stdio.h>**  **void main()**  **{**  **int a,b,c = 30;**  **printf("hello");**  **}**  **24.** **Write a LEX specification count the number of characters, number of lines & number of words.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **int char\_count = 0;**  **int line\_count = 0;**  **int word\_count = 0;**  **int in\_word = 0;**  **%}**  **%%**  **\n { line\_count++; char\_count++; in\_word = 0; }**  **[ \t]+ { char\_count += yyleng; in\_word = 0; }**  **[a-zA-Z0-9]+ { char\_count += yyleng; word\_count++; in\_word = 1; }**  **. { char\_count++; }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc > 1) {**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("Failed to open file");**  **return 1;**  **}**  **yyin = file;**  **}**  **yylex();**  **printf("Number of characters: %d\n", char\_count);**  **printf("Number of lines: %d\n", line\_count);**  **printf("Number of words: %d\n", word\_count);**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **int main(int argc, char \*\*argv) {**  **if (argc != 2) {**  **fprintf(stderr, "Usage: %s <input\_file>\n", argv[0]);**  **return 1;**  **}**  **FILE \*file = fopen(argv[1], "r");**  **if (!file) {**  **perror("Failed to open file");**  **return 1;**  **}**  **int characters = 0;**  **int lines = 0;**  **int words = 0;**  **int in\_word = 0;**  **int c;**  **while ((c = fgetc(file)) != EOF) {**  **characters++;**    **if (c == '\n') {**  **lines++;**  **}**    **if (isspace(c)) {**  **in\_word = 0;**  **} else if (!in\_word) {**  **in\_word = 1;**  **words++;**  **}**  **}**  **fclose(file);**  **printf("Number of characters: %d\n", characters);**  **printf("Number of lines: %d\n", lines);**  **printf("Number of words: %d\n", words);**  **return 0;**  **}** | **Unit -3** |
| **4** | **Day 4** | **UNIT -4**  **-Intermediate Code**  **-** **Assignment Statements**  **-** **Boolean Expressions**  **-** **Backpatching** | **25.** **Write a LEX program to count the frequency of the given word in a given sentence.**  **LEX PRO:**  **%{**  **#include <stdio.h>**  **#include <string.h>**  **char word\_to\_find[100];**  **int word\_count = 0;**  **void count\_word(const char\* str);**  **%}**  **%%**  **{word\_to\_find} { word\_count++; }**  **.|\n { /\* Ignore other characters \*/ }**  **%%**  **int main(int argc, char \*\*argv) {**  **if (argc != 3) {**  **fprintf(stderr, "Usage: %s <word> <sentence>\n", argv[0]);**  **return 1;**  **}**  **strcpy(word\_to\_find, argv[1]);**  **// Concatenate all the remaining arguments to form the sentence**  **for (int i = 2; i < argc; i++) {**  **strcat(word\_to\_find, " ");**  **strcat(word\_to\_find, argv[i]);**  **}**  **// Setting the input string for Flex**  **yy\_scan\_string(argv[2]);**  **yylex();**  **printf("The word '%s' occurs %d times in the given sentence.\n", argv[1], word\_count);**  **return 0;**  **}**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <ctype.h>**  **#define MAX\_LENGTH 1000**  **// Function to convert a string to lowercase**  **void to\_lowercase(char \*str) {**  **for (int i = 0; str[i]; i++) {**  **str[i] = tolower(str[i]);**  **}**  **}**  **// Function to count the frequency of a word in a sentence**  **int count\_word\_frequency(const char \*sentence, const char \*word) {**  **int count = 0;**  **char temp\_sentence[MAX\_LENGTH];**  **char \*token;**    **// Copy the sentence to a temporary buffer and convert to lowercase**  **strncpy(temp\_sentence, sentence, MAX\_LENGTH);**  **temp\_sentence[MAX\_LENGTH - 1] = '\0'; // Ensure null termination**  **to\_lowercase(temp\_sentence);**  **// Convert the word to lowercase**  **char temp\_word[MAX\_LENGTH];**  **strncpy(temp\_word, word, MAX\_LENGTH);**  **temp\_word[MAX\_LENGTH - 1] = '\0'; // Ensure null termination**  **to\_lowercase(temp\_word);**  **// Tokenize the sentence and count occurrences of the word**  **token = strtok(temp\_sentence, " \t\n");**  **while (token != NULL) {**  **if (strcmp(token, temp\_word) == 0) {**  **count++;**  **}**  **token = strtok(NULL, " \t\n");**  **}**  **return count;**  **}**  **int main() {**  **char sentence[MAX\_LENGTH];**  **char word[MAX\_LENGTH];**  **printf("Enter the sentence: ");**  **fgets(sentence, MAX\_LENGTH, stdin);**  **// Remove newline character if present**  **sentence[strcspn(sentence, "\n")] = '\0';**  **printf("Enter the word to count: ");**  **fgets(word, MAX\_LENGTH, stdin);**  **// Remove newline character if present**  **word[strcspn(word, "\n")] = '\0';**  **int frequency = count\_word\_frequency(sentence, word);**  **printf("The word \"%s\" appears %d times in the sentence.\n", word, frequency);**  **return 0;**  **}**  **26.** **The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program.**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <string.h>**  **#define MAX\_LENGTH 100**  **// Function prototypes**  **void process\_identifier(const char \*token);**  **void process\_constant(const char \*token);**  **void process\_operator(const char \*token);**  **int main() {**  **char c;**  **char token[MAX\_LENGTH];**  **int token\_length = 0;**  **// Read and process the input**  **while ((c = getchar()) != EOF) {**  **// Skip redundant spaces, tabs, and newlines**  **if (isspace(c)) {**  **continue;**  **}**  **// Process comments**  **if (c == '/' && (c = getchar()) == '\*') {**  **while ((c = getchar()) != EOF) {**  **if (c == '\*' && (c = getchar()) == '/') {**  **break;**  **}**  **}**  **continue;**  **} else if (c == '/' && c == '/') {**  **while ((c = getchar()) != EOF && c != '\n') {**  **// Skip single-line comments**  **}**  **continue;**  **}**  **// Process tokens**  **token[token\_length++] = c;**  **// If the token length exceeds the max length, print and reset**  **if (token\_length >= MAX\_LENGTH) {**  **token[token\_length] = '\0';**  **printf("Token too long: %s\n", token);**  **token\_length = 0;**  **continue;**  **}**  **if (isspace(c) || c == ';' || c == ',' || c == '(' || c == ')' || c == '{' || c == '}' || c == '+' || c == '-' || c == '\*' || c == '/') {**  **token[token\_length] = '\0';**  **if (token\_length > 0) {**  **// Check if token is an identifier or constant**  **if (isalpha(token[0]) || token[0] == '\_') {**  **process\_identifier(token);**  **} else if (isdigit(token[0])) {**  **process\_constant(token);**  **} else {**  **process\_operator(token);**  **}**  **token\_length = 0;**  **}**  **if (c != ' ' && c != '\t' && c != '\n') {**  **// Print operators and punctuators**  **char op[2] = {c, '\0'};**  **process\_operator(op);**  **}**  **}**  **}**  **// Process any remaining token**  **if (token\_length > 0) {**  **token[token\_length] = '\0';**  **if (isalpha(token[0]) || token[0] == '\_') {**  **process\_identifier(token);**  **} else if (isdigit(token[0])) {**  **process\_constant(token);**  **} else {**  **process\_operator(token);**  **}**  **}**  **return 0;**  **}**  **// Function to process identifiers**  **void process\_identifier(const char \*token) {**  **printf("Identifier: %s\n", token);**  **}**  **// Function to process constants**  **void process\_constant(const char \*token) {**  **printf("Constant: %s\n", token);**  **}**  **// Function to process operators**  **void process\_operator(const char \*token) {**  **printf("Operator: %s\n", token);**  **}**  **27.** **Extend the lexical Analyzer to Check comments, dened as follows in C:**  **a) A comment begins with // and includes all characters until the end of that line.**  **b) A comment begins with /\* and includes all characters through the next occurrence of the character sequence \*/Develop a lexical Analyzer to identify whether a given line is a comment or not.**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <string.h>**  **#define MAX\_LENGTH 1000**  **// Function prototypes**  **void process\_line(const char \*line);**  **void process\_comment(const char \*line, int start, int end);**  **// Function to check if a line is a comment**  **void process\_line(const char \*line) {**  **int len = strlen(line);**  **int i = 0;**  **// Check for single-line comments**  **if (line[i] == '/' && line[i + 1] == '/') {**  **printf("Single-line comment: %s\n", line);**  **return;**  **}**  **// Check for multi-line comments**  **while (i < len) {**  **if (line[i] == '/' && line[i + 1] == '\*') {**  **int start = i;**  **i += 2; // Move past "/\*"**  **while (i < len) {**  **if (line[i] == '\*' && line[i + 1] == '/') {**  **process\_comment(line, start, i + 1);**  **i += 2; // Move past "\*/"**  **break;**  **}**  **i++;**  **}**  **// If no "\*/" found, the comment is not closed**  **if (i >= len) {**  **printf("Unterminated multi-line comment starting at index %d: %s\n", start, line);**  **}**  **} else {**  **i++;**  **}**  **}**  **}**  **// Function to process and print a comment substring**  **void process\_comment(const char \*line, int start, int end) {**  **char comment[MAX\_LENGTH];**  **strncpy(comment, line + start, end - start + 1);**  **comment[end - start + 1] = '\0'; // Null-terminate the substring**  **printf("Multi-line comment: %s\n", comment);**  **}**  **int main() {**  **char line[MAX\_LENGTH];**  **printf("Enter lines of text (Ctrl+D to end input):\n");**  **while (fgets(line, sizeof(line), stdin)) {**  **// Remove newline character from the end if present**  **line[strcspn(line, "\n")] = '\0';**    **// Process each line to check if it's a comment**  **process\_line(line);**  **}**  **return 0;**  **}**  **28.** **Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using regular Arithmetic operators .**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#define MAX\_LENGTH 100**  **// Function to process and print operators**  **void process\_operator(char op) {**  **switch (op) {**  **case '+':**  **printf("Operator: %c (Addition)\n", op);**  **break;**  **case '-':**  **printf("Operator: %c (Subtraction)\n", op);**  **break;**  **case '\*':**  **printf("Operator: %c (Multiplication)\n", op);**  **break;**  **case '/':**  **printf("Operator: %c (Division)\n", op);**  **break;**  **default:**  **printf("Unknown character: %c\n", op);**  **break;**  **}**  **}**  **// Function to tokenize and identify operators**  **void analyze\_input(const char \*input) {**  **for (int i = 0; input[i] != '\0'; i++) {**  **char c = input[i];**  **// Skip whitespace**  **if (isspace(c)) {**  **continue;**  **}**  **// Check if the character is one of the arithmetic operators**  **if (c == '+' || c == '-' || c == '\*' || c == '/') {**  **process\_operator(c);**  **} else {**  **// Print unknown characters**  **printf("Unknown character: %c\n", c);**  **}**  **}**  **}**  **int main() {**  **char input[MAX\_LENGTH];**    **printf("Enter an arithmetic expression: ");**  **if (fgets(input, sizeof(input), stdin)) {**  **// Remove newline character if present**  **//input[strcspn(input, "\n")] = '\0';**  **// Analyze the input**  **analyze\_input(input);**  **} else {**  **fprintf(stderr, "Error reading input.\n");**  **return 1;**  **}**  **return 0;**  **}**  **29.** **Design a lexical Analyzer to find the number of whitespaces and newline characters.**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **int main() {**  **char c;**  **int whitespace\_count = 0;**  **int newline\_count = 0;**  **printf("Enter text (Ctrl+D to end input):\n");**  **while ((c = getchar()) != EOF) {**  **if (isspace(c)) {**  **if (c == ' ' || c == '\t') {**  **whitespace\_count++;**  **} else if (c == '\n') {**  **newline\_count++;**  **}**  **}**  **}**  **printf("Number of whitespace characters: %d\n", whitespace\_count);**  **printf("Number of newline characters: %d\n", newline\_count);**  **return 0;**  **}**  **30.** **Develop a lexical Analyzer to test whether a given identifier is valid or not.**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <string.h>**  **#include <stdbool.h>**  **// List of C keywords**  **const char \*keywords[] = {**  **"auto", "break", "case", "char", "const", "continue", "default", "do",**  **"double", "else", "enum", "extern", "float", "for", "goto", "if", "inline",**  **"int", "long", "register", "restrict", "return", "short", "signed",**  **"sizeof", "static", "struct", "switch", "typedef", "union", "unsigned",**  **"void", "volatile", "while", "\_Bool", "\_Complex", "\_Imaginary"**  **};**  **// Function to check if a given string is a keyword**  **bool is\_keyword(const char \*str) {**  **for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); ++i) {**  **if (strcmp(str, keywords[i]) == 0) {**  **return true;**  **}**  **}**  **return false;**  **}**  **// Function to check if a given identifier is valid**  **bool is\_valid\_identifier(const char \*identifier) {**  **// Check if the first character is a letter or underscore**  **if (!(isalpha(identifier[0]) || identifier[0] == '\_')) {**  **return false;**  **}**  **// Check the remaining characters**  **for (int i = 1; identifier[i] != '\0'; ++i) {**  **if (!(isalnum(identifier[i]) || identifier[i] == '\_')) {**  **return false;**  **}**  **}**  **// Check if the identifier is a keyword**  **if (is\_keyword(identifier)) {**  **return false;**  **}**  **return true;**  **}**  **int main() {**  **char identifier[100];**  **// Prompt user for input**  **printf("Enter an identifier: ");**  **scanf("%s", identifier);**  **// Check if the identifier is valid**  **if (is\_valid\_identifier(identifier)) {**  **printf("\"%s\" is a valid identifier.\n", identifier);**  **} else {**  **printf("\"%s\" is not a valid identifier.\n", identifier);**  **}**  **return 0;**  **}**  **31.Implement a C program to eliminate left recursion.**  **C PRO:**  **#include <stdio.h>**  **#include <string.h>**  **#include <stdlib.h>**  **#include <stdbool.h>**  **#define MAX 100**  **typedef struct {**  **char nonTerminal;**  **char productions[MAX][MAX];**  **int prodCount;**  **} Grammar;**  **void readGrammar(Grammar \*grammar) {**  **printf("Enter the non-terminal: ");**  **scanf(" %c", &grammar->nonTerminal);**  **printf("Enter the number of productions: ");**  **scanf("%d", &grammar->prodCount);**  **printf("Enter the productions (e.g., A -> Ab | c):\n");**  **for (int i = 0; i < grammar->prodCount; i++) {**  **printf("Production %d: ", i + 1);**  **scanf(" %s", grammar->productions[i]);**  **}**  **}**  **void eliminateLeftRecursion(Grammar \*grammar) {**  **char nonTerminal = grammar->nonTerminal;**  **char newNonTerminal = nonTerminal + '1'; // Create a new non-terminal (e.g., if A, then A')**  **char alpha[MAX][MAX], beta[MAX][MAX];**  **int alphaCount = 0, betaCount = 0;**  **for (int i = 0; i < grammar->prodCount; i++) {**  **if (grammar->productions[i][0] == nonTerminal) {**  **strcpy(alpha[alphaCount++], &grammar->productions[i][1]); // A -> Aα**  **} else {**  **strcpy(beta[betaCount++], grammar->productions[i]); // A -> β**  **}**  **}**  **// Print the new grammar without left recursion**  **printf("\nNew Grammar after eliminating left recursion:\n");**  **// Print productions for the original non-terminal**  **printf("%c -> ", nonTerminal);**  **for (int i = 0; i < betaCount; i++) {**  **printf("%s%c", beta[i], newNonTerminal);**  **if (i != betaCount - 1) {**  **printf(" | ");**  **}**  **}**  **printf("\n");**  **// Print productions for the new non-terminal**  **printf("%c -> ", newNonTerminal);**  **for (int i = 0; i < alphaCount; i++) {**  **printf("%s%c", alpha[i], newNonTerminal);**  **if (i != alphaCount - 1) {**  **printf(" | ");**  **}**  **}**  **printf(" | ε\n");**  **}**  **int main() {**  **Grammar grammar;**  **readGrammar(&grammar);**  **eliminateLeftRecursion(&grammar);**  **return 0;**  **}**  **32.** **Implement a C program to eliminate left factoring.**  **#include<stdio.h>**  **#include<string.h>**  **int main()**  **{**  **char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];**  **int i,j=0,k=0,l=0,pos;**  **printf("Enter Production : S->");**  **gets(gram);**  **for(i=0;gram[i]!='|';i++,j++)**  **part1[j]=gram[i];**  **part1[j]='\0';**  **for(j=++i,i=0;gram[j]!='\0';j++,i++)**  **part2[i]=gram[j];**  **part2[i]='\0';**  **for(i=0;i<strlen(part1)||i<strlen(part2);i++)**  **{**  **if(part1[i]==part2[i])**  **{**  **modifiedGram[k]=part1[i];**  **k++;**  **pos=i+1;**  **}**  **}**  **for(i=pos,j=0;part1[i]!='\0';i++,j++){**  **newGram[j]=part1[i];**  **}**  **newGram[j++]='|';**  **for(i=pos;part2[i]!='\0';i++,j++){**  **newGram[j]=part2[i];**  **}**  **modifiedGram[k]='X';**  **modifiedGram[++k]='\0';**  **newGram[j]='\0';**  **printf("\n S->%s",modifiedGram);**  **printf("\n X->%s\n",newGram);**  **}** | **Unit -4** |
| **5** | **DAY 5** | **UNIT- V**  **-Optimization**  **-** **Basic Blocks and Flow Graphs**  **-** **A simple Code generator**  **-** **DAG representation of Basic Blocks** | **33. Implement a C program to perform symbol table operations.**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <string.h>**  **#define TABLE\_SIZE 100**  **typedef struct Symbol {**  **char name[50];**  **char type[20];**  **int scope;**  **struct Symbol\* next;**  **} Symbol;**  **Symbol\* symbolTable[TABLE\_SIZE];**  **unsigned int hash(char\* name) {**  **unsigned int hashValue = 0;**  **for (int i = 0; name[i] != '\0'; i++) {**  **hashValue = hashValue \* 31 + name[i];**  **}**  **return hashValue % TABLE\_SIZE;**  **}**  **void insertSymbol(char\* name, char\* type, int scope) {**  **unsigned int index = hash(name);**  **Symbol\* newSymbol = (Symbol\*)malloc(sizeof(Symbol));**  **strcpy(newSymbol->name, name);**  **strcpy(newSymbol->type, type);**  **newSymbol->scope = scope;**  **newSymbol->next = symbolTable[index];**  **symbolTable[index] = newSymbol;**  **}**  **Symbol\* searchSymbol(char\* name) {**  **unsigned int index = hash(name);**  **Symbol\* current = symbolTable[index];**  **while (current != NULL) {**  **if (strcmp(current->name, name) == 0) {**  **return current;**  **}**  **current = current->next;**  **}**  **return NULL;**  **}**  **void displaySymbolTable() {**  **printf("Symbol Table:\n");**  **printf("Name\tType\tScope\n");**  **printf("-------------------------\n");**  **for (int i = 0; i < TABLE\_SIZE; i++) {**  **Symbol\* current = symbolTable[i];**  **while (current != NULL) {**  **printf("%s\t%s\t%d\n", current->name, current->type, current->scope);**  **current = current->next;**  **}**  **}**  **}**  **int main() {**  **int choice;**  **char name[50], type[20];**  **int scope;**  **while (1) {**  **printf("\nSymbol Table Operations:\n");**  **printf("1. Insert Symbol\n");**  **printf("2. Search Symbol\n");**  **printf("3. Display Symbol Table\n");**  **printf("4. Exit\n");**  **printf("Enter your choice: ");**  **scanf("%d", &choice);**  **switch (choice) {**  **case 1:**  **printf("Enter symbol name: ");**  **scanf("%s", name);**  **printf("Enter symbol type: ");**  **scanf("%s", type);**  **printf("Enter symbol scope: ");**  **scanf("%d", &scope);**  **insertSymbol(name, type, scope);**  **break;**  **case 2:**  **printf("Enter symbol name to search: ");**  **scanf("%s", name);**  **Symbol\*symbol = searchSymbol(name);**  **if (symbol) {**  **printf("Symbol found: %s, Type: %s, Scope: %d\n", symbol->name, symbol->type, symbol->scope);**  **} else {**  **printf("Symbol not found.\n");**  **}**  **break;**    **}**  **}**  **return 0;**  **}**  **34. All languages have Grammar. When people frame a sentence we usually say whether the sentence is framed as per the rules of the Grammar or Not. Similarly use the same ideology , implement to check whether the given input string is satisfying the grammar or not .**  **C PRO:**  **35. Write a C program to construct recursive descent parsing.**  **C PRO:**  **#include <stdio.h>**  **#include <ctype.h>**  **#include <stdbool.h>**  **#include <string.h>**  **const char \*input;**  **int index;**  **bool E();**  **bool E\_prime();**  **bool T();**  **bool T\_prime();**  **bool F();**  **bool match(char expected) {**  **if (input[index] == expected) {**  **index++;**  **return true;**  **}**  **return false;**  **}**  **bool E() {**  **if (T()) {**  **if (E\_prime()) {**  **return true;**  **}**  **}**  **return false;**  **}**  **bool E\_prime() {**  **int savedIndex = index;**  **if (match('+')) {**  **if (T()) {**  **if (E\_prime()) {**  **return true;**  **}**  **}**  **index = savedIndex;**  **return false;**  **}**  **return true; // ε transition**  **}**  **bool T() {**  **if (F()) {**  **if (T\_prime()) {**  **return true;**  **}**  **}**  **return false;**  **}**  **bool T\_prime() {**  **int savedIndex = index;**  **if (match('\*')) {**  **if (F()) {**  **if (T\_prime()) {**  **return true;**  **}**  **}**  **index = savedIndex;**  **return false;**  **}**  **return true; // ε transition**  **}**  **bool F() {**  **int savedIndex = index;**  **if (match('(')) {**  **if (E()) {**  **if (match(')')) {**  **return true;**  **}**  **}**  **index = savedIndex;**  **return false;**  **}**  **if (isalnum(input[index])) { // id (identifier)**  **index++;**  **return true;**  **}**  **return false;**  **}**  **bool parse(const char \*str) {**  **input = str;**  **index = 0;**  **if (E() && input[index] == '\0') {**  **return true;**  **}**  **return false;**  **}**  **int main() {**  **char inputString[100];**  **printf("Enter an arithmetic expression to check: ");**  **scanf("%s", inputString);**  **if (parse(inputString)) {**  **printf("The expression is valid.\n");**  **} else {**  **printf("The expression is invalid.\n");**  **}**  **return 0;**  **}**  **36. In a class of Grade 3, Mathematics Teacher asked for the Acronym PEMDAS?. All of them are thinking for a while. A smart kid of the class Kishore of the class says it is Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. Can you write a C Program to help the students to understand about the operator precedence parsing for an expression containing more than one operator, the order of evaluation depends on the order of operations.**  **C PRO:**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <ctype.h>**  **#include <math.h>**  **typedef enum {NUMBER, OPERATOR, END} TokenType;**  **typedef struct {**  **TokenType type;**  **double value;**  **char op;**  **} Token;**  **const char \*input;**  **int index;**  **Token getToken() {**  **Token token;**  **while (input[index] == ' ') index++; // skip whitespace**  **if (isdigit(input[index]) || input[index] == '.') {**  **token.type = NUMBER;**  **token.value = strtod(&input[index], (char \*\*)&input[index]);**  **} else if (input[index] == '\0') {**  **token.type = END;**  **} else {**  **token.type = OPERATOR;**  **token.op = input[index++];**  **}**  **return token;**  **}**  **double expression();**  **double term();**  **double factor();**  **double power();**  **double primary();**  **double expression() {**  **double result = term();**  **Token token = getToken();**  **while (token.type == OPERATOR && (token.op == '+' || token.op == '-')) {**  **if (token.op == '+') {**  **result += term();**  **} else {**  **result -= term();**  **}**  **token = getToken();**  **}**  **index--;**  **return result;**  **}**  **double term() {**  **double result = factor();**  **Token token = getToken();**  **while (token.type == OPERATOR && (token.op == '\*' || token.op == '/')) {**  **if (token.op == '\*') {**  **result \*= factor();**  **} else {**  **result /= factor();**  **}**  **token = getToken();**  **}**  **index--;**  **return result;**  **}**  **double factor() {**  **double result = power();**  **Token token = getToken();**  **while (token.type == OPERATOR && token.op == '^') {**  **result = pow(result, power());**  **token = getToken();**  **}**  **index--;**  **return result;**  **}**  **double power() {**  **return primary();**  **}**  **double primary() {**  **Token token = getToken();**  **double result;**  **if (token.type == NUMBER) {**  **result = token.value;**  **} else if (token.type == OPERATOR && token.op == '(') {**  **result = expression();**  **token = getToken();**  **if (token.type != OPERATOR || token.op != ')') {**  **printf("Error: Unmatched parentheses\n");**  **exit(1);**  **}**  **} else {**  **printf("Error: Unexpected token\n");**  **exit(1);**  **}**  **return result;**  **}**  **double evaluate(const char \*expr) {**  **input = expr;**  **index = 0;**  **return expression();**  **}**  **int main() {**  **char inputString[100];**  **printf("Enter an arithmetic expression: ");**  **scanf("%[^\n]", inputString);**  **double result = evaluate(inputString);**  **printf("Result: %lf\n", result);**  **return 0;**  **}**  **37. The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to Generate the Three address code representation for the given input statement.**  **C PRO:**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <ctype.h>**  **#include <string.h>**  **#define MAX 100**  **typedef struct {**  **char op;**  **char arg1[MAX];**  **char arg2[MAX];**  **char result[MAX];**  **} ThreeAddressCode;**  **ThreeAddressCode tac[MAX];**  **int tacIndex = 0;**  **const char \*input;**  **int index;**  **void printTAC() {**  **printf("Three Address Code:\n");**  **for (int i = 0; i < tacIndex; i++) {**  **printf("%s = %s %c %s\n", tac[i].result, tac[i].arg1, tac[i].op, tac[i].arg2);**  **}**  **}**  **void addTAC(char op, const char \*arg1, const char \*arg2, const char \*result) {**  **tac[tacIndex].op = op;**  **strcpy(tac[tacIndex].arg1, arg1);**  **strcpy(tac[tacIndex].arg2, arg2);**  **strcpy(tac[tacIndex].result, result);**  **tacIndex++;**  **}**  **char\* newTemp() {**  **static int tempCount = 0;**  **static char temp[10];**  **snprintf(temp, sizeof(temp), "t%d", tempCount++);**  **return temp;**  **}**  **void parseExpression();**  **void parseFactor() {**  **if (isdigit(input[index])) {**  **char temp[10];**  **snprintf(temp, sizeof(temp), "%c", input[index++]);**  **addTAC(' ', temp, " ", newTemp());**  **} else if (input[index] == '(') {**  **index++;**  **parseExpression();**  **if (input[index] == ')') {**  **index++;**  **}**  **}**  **}**  **void parseTerm() {**  **parseFactor();**  **while (input[index] == '\*' || input[index] == '/') {**  **char op = input[index++];**  **parseFactor();**  **char \*result = newTemp();**  **addTAC(op, tac[tacIndex - 2].result, tac[tacIndex - 1].result, result);**  **}**  **}**  **void parseExpression() {**  **parseTerm();**  **while (input[index] == '+' || input[index] == '-') {**  **char op = input[index++];**  **parseTerm();**  **char \*result = newTemp();**  **addTAC(op, tac[tacIndex - 2].result, tac[tacIndex - 1].result, result);**  **}**  **}**  **int main() {**  **char inputString[MAX];**  **printf("Enter an arithmetic expression: ");**  **scanf("%[^\n]", inputString);**  **input = inputString;**  **index = 0;**  **parseExpression();**  **printTAC();**  **return 0;**  **}**  **38. Write a C program for implementing a Lexical Analyzer to Count the number of characters, words, and lines .**  **39. Write a C Program for code optimization to eliminate common subexpression.**  **C PRO:**  **#include <stdio.h>**  **#include <stdlib.h>**  **#include <string.h>**  **#define MAX 100**  **typedef struct {**  **char expression[MAX];**  **char result[MAX];**  **} CodeLine;**  **typedef struct {**  **char expression[MAX];**  **char result[MAX];**  **} CommonSubexpression;**  **CodeLine code[MAX];**  **CommonSubexpression subexpressions[MAX];**  **int codeIndex = 0;**  **int subexprIndex = 0;**  **void addCodeLine(const char \*expr, const char \*result) {**  **strcpy(code[codeIndex].expression, expr);**  **strcpy(code[codeIndex].result, result);**  **codeIndex++;**  **}**  **void addSubexpression(const char \*expr, const char \*result) {**  **strcpy(subexpressions[subexprIndex].expression, expr);**  **strcpy(subexpressions[subexprIndex].result, result);**  **subexprIndex++;**  **}**  **int findSubexpression(const char \*expr) {**  **for (int i = 0; i < subexprIndex; i++) {**  **if (strcmp(subexpressions[i].expression, expr) == 0) {**  **return i;**  **}**  **}**  **return -1;**  **}**  **void optimizeCode() {**  **for (int i = 0; i < codeIndex; i++) {**  **int index = findSubexpression(code[i].expression);**  **if (index != -1) {**  **// Replace with common subexpression result**  **printf("%s = %s\n", code[i].result, subexpressions[index].result);**  **} else {**  **// Add new subexpression**  **char result[MAX];**  **snprintf(result, sizeof(result), "t%d", subexprIndex);**  **addSubexpression(code[i].expression, result);**  **printf("%s = %s\n", result, code[i].expression);**  **}**  **}**  **}**  **int main() {**  **char expr[MAX];**  **char result[MAX];**  **printf("Enter code (type 'end' to finish):\n");**  **while (1) {**  **printf("Expression (e.g., a + b): ");**  **scanf(" %[^\n]", expr);**  **if (strcmp(expr, "end") == 0) {**  **break;**  **}**  **printf("Result variable: ");**  **scanf("%s", result);**  **addCodeLine(expr, result);**  **}**  **printf("\nOptimized Code:\n");**  **optimizeCode();**  **return 0;**  **}**  **40.Write a C program to implement the back end of the compiler.**  **C PRO:**  **#include<stdio.h>**  **#include<conio.h>**  **#include<string.h>**  **int main()**  **{**  **int n,i,j;**  **char a[50][50];**  **printf("enter the no: intermediate code:");**  **scanf("%d",&n);**  **for(i=0;i<n;i++)**  **{**  **printf("enter the 3 address code:%d:",i+1);**  **for(j=0;j<6;j++)**  **{**  **scanf("%c",&a[i][j]);**  **}**  **}**  **printf("the generated code is:");**  **for(i=0;i<n;i++)**  **{**  **printf("\n mov %c,R%d",a[i][3],i);**  **if(a[i][4]=='-')**  **{**  **printf("\n sub %c,R%d",a[i][5],i);**  **}**  **if(a[i][4]=='+')**  **{**  **printf("\n add %c,R%d",a[i][5],i);**  **}**  **if(a[i][4]=='\*')**  **{**  **printf("\n mul %c,R%d",a[i][5],i);**  **}**  **if(a[i][4]=='/')**  **{**  **printf("\n div %c,R%d",a[i][5],i);**  **}**  **printf("\n mov R%d,%c",i,a[i][1]);**  **printf("\n");**  **}**  **return 0;**  **}** | **Unit -5** |