Name: Dhritesh Bhagat
Class Roll: 18
Enrollment No: 12019002002026
Subject Name: Compiler Design Laboratory
Subject Code: PCCCS691
Year: 3rd
Semester: 6
Section: A1

NAME OF THE EXPERIMENT:
Write a C program to identify whether a given line is a comment or not.

Algorithm:

```
Step-1: Start.
Step-2: Create a character type array ‘com’ with length 100 and declare an
integer variable f=0.
Step-3: Taking a line as input from the user.
Step-4: if com[0] = '/'
     if com[1] = '/'
          print "It is a single line comment."
     else if com[1] = '*'
          start a loop from i = 2 to i ⩽ 40 where value of i is
               incremented by 1 i every iteration
               if com[i] = '*' and com[i + 1] = '/'
               print "It is a multi-line comment."
               f = 1
               break
               else
               continue
          if f = 0
               print "It is not a comment."
     else
          print "It is not a comment."
     else
     print "It is not a comment."
Step-5: STOP
```

Source code:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char com[100];
    int i = 2, f = 0;

    printf("\n Enter the line : ");
    gets(com);

    if (com[0] == '/')
    {
        if (com[1] == '/')
            printf("\n It is a single line comment.\n\n");
        else if (com[1] == '*')
        {
            for (i = 2; i <= 40; i++)
            {
                if (com[i] == '*' && com[i + 1] == '/')
                {
                    printf("\n It is a multi-line comment.\n\n");
```

```
                    f = 1;
                    break;
                }
                else
                    continue;
            }
            if (f == 0)
                printf("\n It is not a comment.\n\n");
        }
        else
            printf("\n It is not a comment.\n\n");
    }
    else
        printf("\n It is not a comment.\n\n");
}
```

Output:

```
PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc comment.c -o comment } ; if ($?) { .\comment }

 Enter the line : // Hi, I am Abhirup

 It is a single line comment.

PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc comment.c -o comment } ; if ($?) { .\comment }

 Enter the line : /*My name is Abhirup Dutta.*/

 It is a multi-line comment.

PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc comment.c -o comment } ; if ($?) { .\comment }

 Enter the line : I am Abhirup

 It is not a comment.
```

NAME OF THE EXPERIMENT:
Write a C program to simulate lexical analyzer for validating operators.

Algorithm:
Step-1: START
Step-2: Create a character type array ‘s’ with length 5
Step-3: Take character input from user
Step-4: Perform switch operation on s[0]

```
                    case '>': if(s[1]=='=')
                    print "Greater than or equal."
                    else
                    print "Greater than."
                    break
                    case '<': if(s[1]=='=')
                    print "Less than or equal."
                    else
                    print "Less than."
                    break
                    case '=': if(s[1]=='=')
                    print "Equal to operator."
                    else
                    print "Assignment operator."
                    break
                    case'&': if(s[1]=='&')
                    print "Logical AND."
                    else
                    print "Bitwise AND."
                    break;
                    case'|':if(s[1]=='|')
                    print "Logical OR."
                    else
                    print "Bitwise OR."
                    break;
                  case '!': if(s[1]=='=')
                    print "Not equal."
                    else
                    print "Bitwise Not."
                    break

                    case'+': print "Addition."
                    break
                    case'-':print "Subtraction."
                    break
                    case'*':print "Multiplication."
                    break
                    case'/':print "Division."
                    break
```

```
                    case'%':print "Modulus."
                    break
                    default: print "Not a valid operator!!!"
```

Source code:

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

void main()
{
    char s[5];

    printf("\n Enter any operator : ");

    gets(s);

    switch (s[0])
    {
    case '>':
        if (s[1] == '=')
        {
            printf("\n Greater than or equal.\n\n");
        }
        else
        {
            printf("\n Greater than.\n\n");
        }
        break;

    case '<':
        if (s[1] == '=')
        {
            printf("\n Less than or equal.\n\n");
        }
        else
        {
            printf("\n Less than.\n\n");
        }
        break;

    case '=':
        if (s[1] == '=')
        {
            printf("\n Equal to operator.\n\n");
        }
        else
        {
            printf("\n Assignment operator.\n\n");
        }
        break;

    case '&':
        if (s[1] == '&')
```

```
        {
            printf("\n Logical AND.\n\n");
        }
        else
        {
            printf("\n Bitwise AND.\n\n");
        }
        break;

    case '|':
        if (s[1] == '|')
        {
            printf("\n Logical OR.\n\n");
        }
        else
        {
            printf("\n Bitwise OR.\n\n");
        }
        break;

    case '!':
        if (s[1] == '=')
        {
            printf("\n Not equal.\n\n");
        }
        else
        {
            printf("\n Bitwise Not.\n\n");
        }
        break;

    case '+':
        printf("\n Addition.\n\n");
        break;

    case '-':
        printf("\n Subtraction.\n\n");
        break;

    case '*':
        printf("\n Multiplication.\n\n");
        break;

    case '/':
        printf("\n Division.\n\n");
        break;

    case '%':
        printf("\n Modulus.\n\n");
        break;

    default:
        printf("\n Not a valid operator!!!\n\n");
    }
```

```
}
```

Output:

```
PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc lexical.c -o lexical } ; if ($?) { .\lexical }
 Enter any operator : <=

 Less than or equal.

PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc lexical.c -o lexical } ; if ($?) { .\lexical }

 Enter any operator : =

 Assignment operator.

PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc lexical.c -o lexical } ; if ($?) { .\lexical }

 Enter any operator : +

 Addition.

PS D:\Compiler Design\Lab> cd "d:\Compiler Design\Lab\" ; if ($?) { gcc lexical.c -o lexical } ; if ($?) { .\lexical }

 Enter any operator : 6

 Not a valid operator!!!
```

NAME OF THE EXPERIMENT:
Write a C program to test whether a given identifier is valid or not.

Algorithm:
Step-1: Start.
Step-2: Create a character type array ‘s’ with length 10 and declare an integer variable flag=0.
Step-3: Taking a line as input from the user.
Step-4: if isalpha(s[0])
flag = 1;
else if s[0] == '_'
flag = 1;
else
flag = 0;
if flag == 1
print "\n%s is a valid identifier.\n\n", s;
else
print "\n%s is not a valid identifier.\n\n", s;
Step-5: STOP

Source code:
```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main()
{

    char s[10];
    int flag = 0;

    printf("\nEnter identifier : ");
    gets(s);

    if (isalpha(s[0]))
    {
        flag = 1;
    }
    else if (s[0] == '_')
    {
        flag = 1;
    }
    else
    {
        flag = 0;
```

```
    }

    if (flag == 1)
    {
        printf("\n%s is a valid identifier.\n\n", s);
    }
    else
    {
        printf("\n%s is not a valid identifier.\n\n", s);
    }
}
```

Output:

```
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc identifierVariable.c -o identifierVariable } ; if ($?) { .\identifier
Variable }

Enter identifier : abc


PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc identifierVariable.c -o identifierVariable } ; if ($?) { .\identifier
Variable }

Enter identifier : _abc

_abc is a valid identifier.

PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc identifierVariable.c -o identifierVariable } ; if ($?) { .\identifier
Variable }

Enter identifier : $abc

$abc is not a valid identifier.
```

NAME OF THE EXPERIMENT:
Write a C program to recognize strings under 'a', 'a*b+', 'abb'.

Algorithm:
Step-1: START
Step-2: Create a character type array ‘str’ with length 1000 and declare two integer variables flag = 0 and cntb = 0
Step-3: Take character input from user
Step-4: if str[0] == 'b'
Step-5: We start a loop with integer i = 1, till strlen(str), incrementing i by 1
Step-6: if str[i] == 'a'
flag = 1;
break;
Step-7: End loop
Step-8: else
Step-9: We start a loop with integer i = 1, till strlen(str), incrementing i by 1
Step-10: if str[i] == 'a' && str[i - 1] == 'b'
flag = 1;
break;
Step-11: if str[i] == 'b'
cntb++;
Step-12: End loop
Steo-13: if cntb == 0
flag = 1;
Step-14: if str[0] == 'a' && strlen(str) == 1
print "\n%s falls under the Rule 'a' of recognizing string patterns.\n\n", str;
else if str[0] == 'a' && str[1] == 'b' && str[2] == 'b' && strlen(str) == 3
print "\n%s falls under the Rule 'abb' of recognizing string patterns.\n\n", str;
else if flag == 0
print "\n%s falls under the Rule 'a*b+' of recognizing string patterns.\n\n", str;
else
print "\n%s does not fall under any Rule of recognizing string patterns.\n\n", str;
Step-5: STOP

Source code:
```
#include <stdio.h>
#include <conio.h>
#include <string.h>

int main()
{
    char str[1000];
    int flag = 0, cntb = 0;
```

```
    printf("\nEnter the String : ");
    scanf("%s", str);

    if (str[0] == 'b')
    {
        for (int i = 1; i < strlen(str); i++)
        {
            if (str[i] == 'a')
            {
                flag = 1;
                break;
            }
        }
    }
    else
    {
        for (int i = 1; i < strlen(str); i++)
        {
            if (str[i] == 'a' && str[i - 1] == 'b')
            {
                flag = 1;
                break;
            }

            if (str[i] == 'b')
            {
                cntb++;
            }
        }

        if (cntb == 0)
        {
            flag = 1;
        }
    }

    if (str[0] == 'a' && strlen(str) == 1)
    {
        printf("\n%s falls under the Rule 'a' of recognizing string patterns.\n\n", str);
    }
    else if (str[0] == 'a' && str[1] == 'b' && str[2] == 'b' && strlen(str) == 3)
    {
        printf("\n%s falls under the Rule 'abb' of recognizing string patterns.\n\n",
str);
    }
    else if (flag == 0)
    {
        printf("\n%s falls under the Rule 'a*b+' of recognizing string patterns.\n\n",
str);
    }
    else
    {
        printf("\n%s does not fall under any Rule of recognizing string patterns.\n\n",
str);
```

```
    }
    return 0;
}
```

Output:

```
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc patternMatch.c -o patternMatch } ; if ($?) { .\patternMatch }
Enter the String : b
b falls under the Rule 'a*b+' of recognizing string patterns.
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc patternMatch.c -o patternMatch } ; if ($?) { .\patternMatch }
Enter the String : aab
aab falls under the Rule 'a*b+' of recognizing string patterns.
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc patternMatch.c -o patternMatch } ; if ($?) { .\patternMatch }
Enter the String : aa
aa does not fall under any Rule of recognizing string patterns.
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc patternMatch.c -o patternMatch } ; if ($?) { .\patternMatch }
Enter the String : abb
abb falls under the Rule 'abb' of recognizing string patterns.
PS D:\Compiler Design\Lab\Day 2> cd "d:\Compiler Design\Lab\Day 2\" ; if ($?) { gcc patternMatch.c -o patternMatch } ; if ($?) { .\patternMatch }
Enter the String : a
a falls under the Rule 'a' of recognizing string patterns.
```

NAME OF THE EXPERIMENT:
Write a program on C to build a lexical analyzer which will be able to identify and categorize identifiers, constants, keywords, special characters, operators etc from a C program given as input.

Algorithm:
Step-1: Start.
Step-2: Create a function isDelimiter(char ch) that takes character input to mark delimiter i.e. end of statement.
Step-3: Create a function isOperator(char ch) that takes character input to check whether it is an operator or not.
Step-4: Create a function validIdentifier(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not start with a number.
Step-5: Create a function isKeyword(char *str) that takes string input to check whether it is a keyword or not.
Step-6: Create a function isInteger(char *str) that takes string input to check whether it is an integer or not.
Step 7: Create a function isRealNumber(char *str) that takes string input to check whether it is a real number or not.
Step-8: Create a function *trim(char *s) that takes a string as input and trims the extra white spaces from start and end of the string.
Step-9: Create a function isComment(char *str) that takes string input and checks whether the string starts with (‘//’) or (‘/*’ and ends with ‘*/’).
Step-10: Create a function isHeader(char *str) that takes string input and checks whether the string ends with ‘.h’ making it a header file.
Step-11: Create a function isSpecial(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not contain a special character.
Step-12: Create a function isFormat(char *str) that takes a string input to check whether it is a format specifier or not based on whether it is of the pattern “%d” or “%ld”, etc.
Step-13: Create a function *subString(char *str, int left, int right) that takes a string, and 2 indexes left and right as input to extract the substring form a larger string str.
Step-14: Create a function parse(char *str) that takes a large string as input. Then declare a file pointer *fcode. Then open a file temporary “codefile.txt” that is pointed by fcode.
Step-15: Then check with the boolear function isComment(str) if the string is a comment or not.
Step-16: If the string is not a comment line, the we start checking each string based on index extraction of strings from left and right, thereby checking isDelimiter(str[right]), then increment right by 1.
Step-17: if isDelimiter(str[right]) == true && left == right, then we check if isOperator(str[right]) == true, to store in the file that the character is an operator.

Step-18: else if isDelimiter(str[right]) == true && left ≠ right || (right == len && left ≠ right), then char *subStr = subString(str, left, right - 1)
Step-19: Then we check whether the substring is a keyword or header file or integer or real number or format specifier or contains special character or is a valid identifier based on the boolean functions we created above and save the corresponding message in the file.
Step-20: Then we make left = right and close the file finally with fclose(fcode).
Step-21: Within the main function, we ask the user to enter the code and terminate entering by pressing “Ctrl+Z” followed by “Enter”.
Step-22: We create another file pointer FILE *fptr.
Step-23: As the user enters code line by line, we use a counter to count the number of lines in the program input by the user.
Step-24: Then we print the lexical analysis of the input code by taking the first string from the file str = fgetc(fptr).
Step-25: While we do not reach the end of file, we print the content of the file.
Step-26: Then we close the file with fclose(fptr) and print the total number of lines in the program.
Step-27: Finally, we remove the temporary file with remove("codefile.txt").
Step-28: STOP

Source code:

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

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char *str)
{
```

```
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char *str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") ||
!strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str,
"case") || !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") ||
!strcmp(str, "unsigned") || !strcmp(str, "void") || !strcmp(str, "static") ||
!strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "#include") ||
!strcmp(str, "main") || !strcmp(str, "printf"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char *str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
|| (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char *str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
```

```
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
&& str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Returns string after removing extra white spaces
char *trim(char *s)
{
    int i;

    while (isspace(*s))
        s++;
    for (i = strlen(s) - 1; (isspace(s[i])); i--)
        ;
    s[i + 1] = '\0';
    return s;
}

// Returns 'true' if the string is a COMMENT LINE.
bool isComment(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);
    if (length < 2)
        return false;
    else if ((str[0] == str[1]) && (str[0] == '/'))
        return true;
    else if ((str[0] == str[length - 1]) && (str[1] == str[length - 2]))
    {
        if (str[0] == '/' && str[1] == '*')
            return true;
    }
    else
        return false;
}

// Returns 'true' if the string is a HEADER FILE.
bool isHeader(char *str)
{
    int length = strlen(str);

    if ((str[length - 1] == 'h') && (str[length - 2] == '.'))
    {
        return true;
    }

    return false;
}
```

```
// Returns 'true' if the string contains a special charecter.
bool isSpecial(char *str)
{
    int length = strlen(str), f = 0;

    for (int i = 0; i < length; i++)
    {
        char ch = str[i];
        if (ch == '_')
            f = 0;

        else if (((int)ch >= 32 && (int)ch <= 47) || ((int)ch >= 58 && (int)ch <= 64) ||
((int)ch >= 91 && (int)ch <= 96) || ((int)ch >= 123 && (int)ch <= 127))
        {
            f = 1;
            break;
        }
        else
        {
            f = 0;
        }
    }

    if (f == 1)
        return true;

    return false;
}

// Returns true if the string is a format specifier
bool isFormat(char *str)
{
    int length = strlen(str);

    if ((length <= 5) && (str[1] == '%'))
        return true;

    return false;
}

// Extracts the SUBSTRING.
char *subString(char *str, int left, int right)
{
    int i;
    char *subStr = (char *)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}
```

```
// Parsing the input STRING.
void parse(char *str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    FILE *fcode;

    fcode = fopen("codefile.txt", "a");

    if (isComment(str))
    {
        fprintf(fcode, "'%s' IS A COMMENT LINE.\n", str);
    }
    else
    {
        while (right <= len && left <= right)
        {
            if (isDelimiter(str[right]) == false)
                right++;

            if (isDelimiter(str[right]) == true && left == right)
            {
                if (isOperator(str[right]) == true)
                    fprintf(fcode, "'%c' IS AN OPERATOR.\n", str[right]);

                right++;
                left = right;
            }
            else if (isDelimiter(str[right]) == true && left != right || (right == len &&
left != right))
            {
                char *subStr = subString(str, left, right - 1);

                if (isKeyword(subStr) == true)
                    fprintf(fcode, "'%s' IS A KEYWORD.\n", subStr);

                else if (isHeader(subStr) == true)
                    fprintf(fcode, "'%s' IS A HEADER FILE.\n", subStr);

                else if (isInteger(subStr) == true)
                    fprintf(fcode, "'%s' IS AN INTEGER (CONSTANT).\n", subStr);

                else if (isRealNumber(subStr) == true)
                    fprintf(fcode, "'%s' IS A REAL NUMBER.\n", subStr);

                else if (isFormat(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                    fprintf(fcode, "'%s' IS A FORMAT SPECIFIER.\n", subStr);

                else if (isSpecial(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                    fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT CONTAINS A
SPECIAL CHARACTER.\n", subStr);
```

```
                else if (validIdentifier(subStr) == true && isDelimiter(str[right - 1])
== false)
                    fprintf(fcode, "'%s' IS A VALID IDENTIFIER.\n", subStr);

                else if (validIdentifier(subStr) == false && isDelimiter(str[right - 1])
== false)
                    fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT STARTS WITH
A NUMBER.\n", subStr);

                left = right;
            }
        }
    }

    fclose(fcode);

    return;
}

// DRIVER FUNCTION
int main()
{
    char s[5000];
    int temp, cnt = 0;
    FILE *fptr;
    char str;

    printf("================================================================================");
    printf("\n\tOn Completion of code, press (Ctrl+Z) followed by (Enter)\n");
    printf("================================================================================");

    printf("\nEnter your code here : \n");

    while (1)
    {
        temp = scanf("%[^\n]%*c", s);

        if (temp == -1)
        {
            break;
        }
        else
        {
            parse(s);

            cnt++;
        }
    }

    fptr = fopen("codefile.txt", "r");
    printf("\n\nThe lexical analysis of the code is : \n\n");
    str = fgetc(fptr);
    while (str != EOF)
```

```
    {
        printf("%c", str);
        str = fgetc(fptr);
    }
    printf("\n");
    fclose(fptr);

    printf("Total number of lines in program : %d\n\n", cnt);

    remove("codefile.txt");

    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 3> cd "d:\3rd Year
eLexicalAnalyzer.c -o completeLexicalAnalyzer } ; if ($?) { .\completeLexicalAna
================================================================================
        On Completion of code, press (Ctrl+Z) followed by (Enter)
================================================================================
Enter your code here :
#include <stdio.h>
// Program to add two numbers
int main()
{
int a = 5;
int c = 10;
int 8var = 9;
int var$c = 15;
int d = a + b;
printf("%d",d);
return 0;
}
^Z
```

```
The lexical analysis of the code is :

'#include' IS A KEYWORD.
'<' IS AN OPERATOR.
'stdio.h' IS A HEADER FILE.
'>' IS AN OPERATOR.
'// Program to add two numbers' IS A COMMENT LINE.
'int' IS A KEYWORD.
'main' IS A KEYWORD.
'int' IS A KEYWORD.
'a' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'5' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'c' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'10' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'8var' IS NOT A VALID IDENTIFIER BECAUSE IT STARTS WITH A NUMBER.
'=' IS AN OPERATOR.
'9' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'var$c' IS NOT A VALID IDENTIFIER BECAUSE IT CONTAINS A SPECIAL CHARACTER.
'=' IS AN OPERATOR.
'15' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'd' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'a' IS A VALID IDENTIFIER.
'+' IS AN OPERATOR.
'b' IS A VALID IDENTIFIER.
'printf' IS A KEYWORD.
'"%d"' IS A FORMAT SPECIFIER.
'd' IS A VALID IDENTIFIER.
'return' IS A KEYWORD.
'0' IS AN INTEGER (CONSTANT).

Total number of lines in program : 12
```

NAME OF THE EXPERIMENT:
Create a symbol table in C language. Mention the data structure used and take the inputs from files.

Algorithm:
Step-1: Start.
Step-2: Create a user specific data structure called “symbolTab” that is used to contain 2 string variables representing the data type and variable name respectively along with an integer variable for the address allocation of the variable.
Step-3: Then we take two variables “first” and “last” to specify the first and last indexes of the symbol table along with a global variable size to specify the number of variables allotted in the program.
Step-4: Allocate a variable address to specify the starting address to allocate variables with respect to particular data types.
Step-5: Create two file pointers one for writing and another for reading an error logs file that were observed during compilation of the code.
Ste-6: Create a search function, that will check whether a variable received from the parser is already present within the symbol table or not.
Step-7: Create a function isKeywordVar to check is the variable is of the mention data types being either “int”, “float”, “long”, etc.
Step-8: Create a function insert that will check if the variable name is already present in the symbol table or not. If present, then it will place an error message within the errorLogs file else will insert the variable name with its data type and address within the user defined data type symbolTab.
Step-9: Create a function “display” that will display the entire symbol table from the first pointer till the last created after reading the entire code entered by the user.
Step-10: Create a function isDelimiter(char ch) that takes character input to mark delimiter i.e. end of statement.
Step-11: Create a function isOperator(char ch) that takes character input to check whether it is an operator or not.
Step-12: Create a function validIdentifier(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not start with a number.
Step-13: Create a function isKeyword(char *str) that takes string input to check whether it is a keyword or not.
Step-14: Create a function isInteger(char *str) that takes string input to check whether it is an integer or not.
Step 15: Create a function isRealNumber(char *str) that takes string input to check whether it is a real number or not.
Step-16: Create a function *trim(char *s) that takes a string as input and trims the extra white spaces from start and end of the string.
Step-17: Create a function isComment(char *str) that takes string input and checks whether the string starts with (‘//’) or (‘/*’ and ends with ‘*/’).
Step-18: Create a function isHeader(char *str) that takes string input and checks whether the string ends with ‘.h’ making it a header file.

Step-19: Create a function isSpecial(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not contain a special character.
Step-20: Create a function isFormat(char *str) that takes a string input to check whether it is a format specifier or not based on whether it is of the pattern “%d” or “%ld”, etc.
Step-21: Create a function *subString(char *str, int left, int right) that takes a string, and 2 indexes left and right as input to extract the substring form a larger string str.
Step-22: Create a function parse(char *str) that takes a large string as input. Then declare a file pointer *fcode. Then open a file temporary “codefile.txt” that is pointed by fcode.
Step-23: Then check with the boolear function isComment(str) if the string is a comment or not.
Step-24: If the string is not a comment line, the we start checking each string based on index extraction of strings from left and right, thereby checking isDelimiter(str[right]), then increment right by 1.
Step-25: if isDelimiter(str[right]) == true && left == right, then we check if isOperator(str[right]) == true, to store in the file that the character is an operator.
Step-26: else if isDelimiter(str[right]) == true && left ≠ right || (right == len && left ≠ right), then char *subStr = subString(str, left, right - 1)
Step-27: Then we check whether the substring is a keyword or header file or integer or real number or format specifier or contains special character or is a valid identifier based on the boolean functions we created above and save the corresponding message in the file.
Step-28: If the variable is a keyword, we copy it into a temporary variable.
Step-29: If we find a valid identifier, then check if it is preceded by a data type specified earlier. If it is specied, then we pass the variable to our insert function else we place a message into the error log file.
Step-30: Then we make left = right and close the file finally with fclose(fcode).
Step-31: Within the main function, we ask the user to enter the code and terminate entering by pressing “Ctrl+Z” followed by “Enter”.
Step-32: We create another file pointer FILE *fptr.
Step-33: As the user enters code line by line, we use a counter to count the number of lines in the program input by the user.
Step-34: Then we print the lexical analysis of the input code by taking the first string from the file str = fgetc(fptr).
Step-35: While we do not reach the end of file, we print the content of the file.
Step-36: Then we close the file with fclose(fptr) and print the total number of lines in the program.
Step-37: We then print the entire symbol table with the display function mentioned above.
Step-38: We start reading the error logs file to print any error (if any) within the code we passed into our program and print them.
Step-39: Finally, we remove both the temporary files with remove("codefile.txt") and remove("errorLogs.txt") respectively.

Step-40: STOP

Source code:

```
#include <stdbool.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define null 0

// Symbol table created
struct symbolTab
{
    char dataType[20];
    char varName[20];
    int address;
    struct symbolTab *next;
};

// First and last pointers of symbol table
struct symbolTab *first, *last;

// Initially the size is zero
int size = 0;

// Initialize the primary address to allocate the variables to memory
int addr = 1000;

// File to note the errors in the code
FILE *errorLogs, *errorLogsRead;

// Searches for a particular variable name to avoid duplication
int search(char var[])
{
    int i, flag = 0;
    struct symbolTab *p;
    p = first;
    for (i = 0; i < size; i++)
    {
        if (strcmp(p->varName, var) == 0)
        {
            flag = 1;
        }
        p = p->next;
    }
    return flag;
}

// Returns 'true' if the string is a Keyword representing a variable
bool isKeywordVar(char *str)
{
```

```
    if (!strcmp(str, "int") || !strcmp(str, "float") || !strcmp(str, "char") ||
!strcmp(str, "double") || !strcmp(str, "long"))
        return true;

    return false;
}

// Function to insert a variable into symbol table
void insert(char *str, char *datType)
{
    int n;
    n = search(str);

    errorLogs = fopen("errorLogs.txt", "a");

    if (n == 1)
    {
        fprintf(errorLogs, "The variable name %s already exists, hence the variable is no
more inserted into symbol table.\n", str);
    }
    else
    {
        if (isKeywordVar(datType))
        {
            struct symbolTab *p;
            p = malloc(sizeof(struct symbolTab));
            strcpy(p->dataType, datType);
            strcpy(p->varName, str);
            p->address = addr++;
            p->next = null;
            if (size == 0)
            {
                first = p;
                last = p;
            }
            else
            {
                last->next = p;
                last = p;
            }
            size++;
        }
        else
        {
            fprintf(errorLogs, "No data type is mentioned for the variable %s.\n", str);
        }
    }
    fclose(errorLogs);
}

// Displays the final symbol table
void display()
{
    int i;
```

```
    struct symbolTab *p;
    p = first;
    printf("SRL NO.\t\tDATA TYPE\tLABEL\t\tADDRESS\n");
    for (i = 0; i < size; i++)
    {
        printf("%d\t\t%s\t\t%s\t\t%d\n", i + 1, p->dataType, p->varName, p->address);
        p = p->next;
    }
}

// Returns string after removing extra white spaces
char *trim(char *s)
{
    int i;

    while (isspace(*s))
        s++;
    for (i = strlen(s) - 1; (isspace(s[i])); i--)
        ;
    s[i + 1] = '\0';
    return s;
}

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char *str)
{
    strcpy(str, trim(str));
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
```

```
    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char *str)
{
    strcpy(str, trim(str));
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") ||
!strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str,
"case") || !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") ||
!strcmp(str, "unsigned") || !strcmp(str, "void") || !strcmp(str, "static") ||
!strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "#include") ||
!strcmp(str, "main") || !strcmp(str, "printf"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
|| (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
&& str[i] != '.' ||
            (str[i] == '-' && i > 0))
```

```
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Returns 'true' if the string is a COMMENT LINE.
bool isComment(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);
    if (length < 2)
        return false;
    else if ((str[0] == str[1]) && (str[0] == '/'))
        return true;
    else if ((str[0] == str[length - 1]) && (str[1] == str[length - 2]))
    {
        if (str[0] == '/' && str[1] == '*')
            return true;
    }
    else
        return false;
}

// Returns 'true' if the string is a HEADER FILE.
bool isHeader(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);

    if ((str[length - 1] == 'h') && (str[length - 2] == '.'))
    {
        return true;
    }

    return false;
}

// Returns 'true' if the string contains a special charecter.
bool isSpecial(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str), f = 0;

    for (int i = 0; i < length; i++)
    {
        char ch = str[i];
        if (ch == '_')
            f = 0;

        else if (((int)ch >= 32 && (int)ch <= 47) || ((int)ch >= 58 && (int)ch <= 64) ||
((int)ch >= 91 && (int)ch <= 96) || ((int)ch >= 123 && (int)ch <= 127))
        {
```

```
            f = 1;
            break;
        }
        else
        {
            f = 0;
        }
    }

    if (f == 1)
        return true;

    return false;
}

// Returns true if the string is a format specifier
bool isFormat(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);

    if ((length <= 5) && (str[1] == '%'))
        return true;

    return false;
}

// Extracts the SUBSTRING.
char *subString(char *str, int left, int right)
{
    strcpy(str, trim(str));
    int i;
    char *subStr = (char *)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char *str)
{
    int left = 0, right = 0, flag;
    int len = strlen(str);
    char tempStr[30];

    FILE *fcode;

    fcode = fopen("codefile.txt", "a");

    if (isComment(str))
    {
```

```
        fprintf(fcode, "'%s' IS A COMMENT LINE.\n", str);
    }
    else
    {
        while (right <= len && left <= right)
        {
            if (isDelimiter(str[right]) == false)
                right++;

            if (isDelimiter(str[right]) == true && left == right)
            {
                if (isOperator(str[right]) == true)
                    fprintf(fcode, "'%c' IS AN OPERATOR.\n", str[right]);

                right++;
                left = right;
            }
            else if (isDelimiter(str[right]) == true && left != right || (right == len &&
left != right))
            {
                char *subStr = subString(str, left, right - 1);

                if (strcmp(subStr, ""))
                {

                    if (isKeyword(subStr) == true)
                    {
                        fprintf(fcode, "'%s' IS A KEYWORD.\n", subStr);

                        strcpy(tempStr, subStr);
                    }

                    else if (isHeader(subStr) == true)
                        fprintf(fcode, "'%s' IS A HEADER FILE.\n", subStr);

                    else if (isInteger(subStr) == true)
                        fprintf(fcode, "'%s' IS AN INTEGER (CONSTANT).\n", subStr);

                    else if (isRealNumber(subStr) == true)
                        fprintf(fcode, "'%s' IS A REAL NUMBER.\n", subStr);

                    else if (isFormat(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                        fprintf(fcode, "'%s' IS A FORMAT SPECIFIER.\n", subStr);

                    else if (isSpecial(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                        fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT
CONTAINS A SPECIAL CHARACTER.\n", subStr);

                    else if (validIdentifier(subStr) == true && isDelimiter(str[right -
1]) == false)
                    {
                        fprintf(fcode, "'%s' IS A VALID IDENTIFIER.\n", subStr);
```

```
                        insert(subStr, tempStr);
                    }

                    else if (validIdentifier(subStr) == false && isDelimiter(str[right -
1]) == false)
                        fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT EITHER
STARTS WITH A NUMBER.\n", subStr);

                    left = right;
                }
            }
        }
    }

    fclose(fcode);

    return;
}

// DRIVER FUNCTION
int main()
{
    char s[5000];
    int temp, cnt = 0;
    FILE *fptr;
    char str, strTwo;

    printf("================================================================================");
    printf("\n\tOn Completion of code, press (Ctrl+Z) followed by (Enter)\n");
    printf("================================================================================");

    printf("\nEnter your code here : \n");

    while (1)
    {
        temp = scanf("%[^\n]%*c", s);

        if (temp == -1)
        {
            break;
        }
        else
        {
            parse(s);

            cnt++;
        }
    }

    fptr = fopen("codefile.txt", "r");
    printf("\n\nThe lexical analysis of the code is : \n\n");
    str = fgetc(fptr);
    while (str != EOF)
```

```
    {
        printf("%c", str);
        str = fgetc(fptr);
    }
    printf("\n");
    fclose(fptr);

    printf("Total number of lines in program : %d\n\n", cnt);

    printf("\nFinal Symbol table : \n\n");

    display();

    printf("\n");

    printf("\nError Logs : \n\n");

    errorLogsRead = fopen("errorLogs.txt", "r");
    strTwo = fgetc(errorLogsRead);
    while (strTwo != EOF)
    {
        printf("%c", strTwo);
        strTwo = fgetc(errorLogsRead);
    }
    printf("\n\n");

    fclose(errorLogsRead);

    remove("codefile.txt");
    remove("errorLogs.txt");

    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 4> cd "d:\3rd
able } ; if ($?) { .\symbolTable }
================================================================================
        On Completion of code, press (Ctrl+Z) followed by (Enter)
================================================================================
Enter your code here :
#include <stdio.h>
// Program to add two numbers
void main()
{
char chr;
int a = 5;
int b = 10;
f;
int 8var = 9;
int var$c = 15;
int d = a + b;
printf("%d",d);
}
^Z
```

```
The lexical analysis of the code is :

'#include' IS A KEYWORD.
'<' IS AN OPERATOR.
'stdio.h' IS A HEADER FILE.
'>' IS AN OPERATOR.
'// Program to add two numbers' IS A COMMENT LINE.
'void' IS A KEYWORD.
'main' IS A KEYWORD.
'char' IS A KEYWORD.
'chr' IS A VALID IDENTIFIER.
'int' IS A KEYWORD.
'a' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'5' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'b' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'10' IS AN INTEGER (CONSTANT).
'f' IS A VALID IDENTIFIER.
'int' IS A KEYWORD.
'8var' IS NOT A VALID IDENTIFIER BECAUSE IT EITHER STARTS WITH A NUMBER.
'=' IS AN OPERATOR.
'9' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'var$c' IS NOT A VALID IDENTIFIER BECAUSE IT CONTAINS A SPECIAL CHARACTER.
'=' IS AN OPERATOR.
'15' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'd' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'a' IS A VALID IDENTIFIER.
'+' IS AN OPERATOR.
'b' IS A VALID IDENTIFIER.
'printf' IS A KEYWORD.
'"%d"' IS A FORMAT SPECIFIER.
'd' IS A VALID IDENTIFIER.

Total number of lines in program : 13


Final Symbol table :

SRL NO.         DATA TYPE       LABEL           ADDRESS
1               char            chr             1000
2               int             a               1001
3               int             b               1002
4               int             d               1003


Error Logs :

No data type is mentioned for the variable f.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name b already exists, hence the variable is no more inserted into symbol table.
The variable name d already exists, hence the variable is no more inserted into symbol table.
```

NAME OF THE EXPERIMENT:
Write a C program to find spelling mistakes for inbuilt functions in a C program.

Algorithm:
Step-1: Start.
Step-2: Create a user specific data structure called “symbolTab” that is used to contain 2 string variables representing the data type and variable name respectively along with an integer variable for the address allocation of the variable.
Step-3: Then we take two variables “first” and “last” to specify the first and last indexes of the symbol table along with a global variable size to specify the number of variables allotted in the program.
Step-4: Allocate a variable address to specify the starting address to allocate variables with respect to particular data types.
Step-5: Create two file pointers one for writing and another for reading an error logs file that were observed during compilation of the code.
Ste-6: Create a search function, that will check whether a variable received from the parser is already present within the symbol table or not.
Step-7: Create a function isKeywordVar to check is the variable is of the mention data types being either “int”, “float”, “long”, etc.
Step-8: Create a function insert that will check if the variable name is already present in the symbol table or not. If present, then it will place an error message within the errorLogs file else will insert the variable name with its data type and address within the user defined data type symbolTab.
Step-9: Create a function “display” that will display the entire symbol table from the first pointer till the last created after reading the entire code entered by the user.
Step-10: Create a function isDelimiter(char ch) that takes character input to mark delimiter i.e. end of statement.
Step-11: Create a function isOperator(char ch) that takes character input to check whether it is an operator or not.
Step-12: Create a function validIdentifier(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not start with a number.
Step-13: Create a function isKeyword(char *str) that takes string input to check whether it is a keyword or not.
Step-14: Create a function isInteger(char *str) that takes string input to check whether it is an integer or not.
Step 15: Create a function isRealNumber(char *str) that takes string input to check whether it is a real number or not.
Step-16: Create a function *trim(char *s) that takes a string as input and trims the extra white spaces from start and end of the string.
Step-17: Create a function isComment(char *str) that takes string input and checks whether the string starts with (‘//’) or (‘/*’ and ends with ‘*/’).
Step-18: Create a function isHeader(char *str) that takes string input and checks whether the string ends with ‘.h’ making it a header file.

Step-19: Create a function isSpecial(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not contain a special character.
Step-20: Create a function isFormat(char *str) that takes a string input to check whether it is a format specifier or not based on whether it is of the pattern “%d” or “%ld”, etc.
Step-21: Create a function isSpellError(char *str) that takes string input to check whether it is spelling mistake with respect to keywords like “if”, “for”, etc.
Step-22: Create a function *subString(char *str, int left, int right) that takes a string, and 2 indexes left and right as input to extract the substring form a larger string str.
Step-23: Create a function parse(char *str) that takes a large string as input. Then declare a file pointer *fcode. Then open a file temporary “codefile.txt” that is pointed by fcode.
Step-24: Then check with the boolear function isComment(str) if the string is a comment or not.
Step-25: If the string is not a comment line, the we start checking each string based on index extraction of strings from left and right, thereby checking isDelimiter(str[right]), then increment right by 1.
Step-26: if isDelimiter(str[right]) == true && left == right, then we check if isOperator(str[right]) == true, to store in the file that the character is an operator.
Step-27: else if isDelimiter(str[right]) == true && left ≠ right || (right == len && left ≠ right), then char *subStr = subString(str, left, right - 1)
Step-28: We check if delimiter character is a ‘{‘, then we check if the preceding string has spelling mistake with respect to words “do” or else.
Step-29: We check if the delimiter character is a ‘(‘, then we check if the preceding string has spelling mistake with respect to words “if” or “for” or “while” or “scanf” or “printf” or “gets” or “getch” or “main” via the function isSpellError(char *str).
Step-30: Then we check whether the substring is a keyword or header file or integer or real number or format specifier or contains special character or is a valid identifier based on the boolean functions we created above and save the corresponding message in the file.
Step-31: If the variable is a keyword, we copy it into a temporary variable.
Step-32: If we find a valid identifier, then check if it is preceded by a data type specified earlier. If it is specied, then we pass the variable to our insert function else we place a message into the error log file.
Step-33: Then we make left = right and close the file finally with fclose(fcode).
Step-34: Within the main function, we ask the user to enter the code and terminate entering by pressing “Ctrl+Z” followed by “Enter”.
Step-35: We create another file pointer FILE *fptr.
Step-36: As the user enters code line by line, we use a counter to count the number of lines in the program input by the user.
Step-37: Then we print the lexical analysis of the input code by taking the first string from the file str = fgetc(fptr).

Step-38: While we do not reach the end of file, we print the content of the file.
Step-39: Then we close the file with fclose(fptr) and print the total number of lines in the program.
Step-40: We then print the entire symbol table with the display function mentioned above.
Step-41: We start reading the error logs file to print any error (if any) within the code we passed into our program and print them.
Step-42: We start reading the spellCheck file to print any spelling errors (if any) within the code we passed into our program and print them.
Step-43: Finally, we remove both the temporary files with remove("codefile.txt"), remove("errorLogs.txt") and remove("spellCheck.txt") respectively.
Step-44: STOP

Source code:

```
#include <stdbool.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define null 0

// Symbol table created
struct symbolTab
{
    char dataType[20];
    char varName[20];
    int address;
    struct symbolTab *next;
};

// First and last pointers of symbol table
struct symbolTab *first, *last;

// Initially the size is zero
int size = 0;

// Initialize the primary address to allocate the variables to memory
int addr = 1000;

// File to note the errors in the code
FILE *errorLogs, *errorLogsRead;

// Searches for a particular variable name to avoid duplication
int search(char var[])
{
    int i, flag = 0;
    struct symbolTab *p;
```

```
    p = first;
    for (i = 0; i < size; i++)
    {
        if (strcmp(p->varName, var) == 0)
        {
            flag = 1;
        }
        p = p->next;
    }
    return flag;
}

// Returns 'true' if the string is a Keyword representing a variable
bool isKeywordVar(char *str)
{
    if (!strcmp(str, "int") || !strcmp(str, "float") || !strcmp(str, "char") ||
!strcmp(str, "double") || !strcmp(str, "long"))
        return true;

    return false;
}

// Function to insert a variable into symbol table
void insert(char *str, char *datType)
{
    int n;
    n = search(str);

    errorLogs = fopen("errorLogs.txt", "a");

    if (n == 1)
    {
        fprintf(errorLogs, "The variable name %s already exists, hence the variable is no
more inserted into symbol table.\n", str);
    }
    else
    {
        if (isKeywordVar(datType))
        {
            struct symbolTab *p;
            p = malloc(sizeof(struct symbolTab));
            strcpy(p->dataType, datType);
            strcpy(p->varName, str);
            p->address = addr++;
            p->next = null;
            if (size == 0)
            {
```

```
                first = p;
                last = p;
            }
            else
            {
                last->next = p;
                last = p;
            }
            size++;
        }
        else
        {
            fprintf(errorLogs, "No data type is mentioned for the variable %s.\n", str);
        }
    }
    fclose(errorLogs);
}

// Displays the final symbol table
void display()
{
    int i;
    struct symbolTab *p;
    p = first;
    printf("SRL NO.\t\tDATA TYPE\tLABEL\t\tADDRESS\n");
    for (i = 0; i < size; i++)
    {
        printf("%d\t\t%s\t\t%s\t\t%d\n", i + 1, p->dataType, p->varName, p->address);
        p = p->next;
    }
}

// Returns string after removing extra white spaces
char *trim(char *s)
{
    int i;

    while (isspace(*s))
        s++;
    for (i = strlen(s) - 1; (isspace(s[i])); i--)
        ;
    s[i + 1] = '\0';
    return s;
}

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
```

```
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char *str)
{
    strcpy(str, trim(str));
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);

    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char *str)
{
    strcpy(str, trim(str));
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") ||
!strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str,
"case") || !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") ||
!strcmp(str, "unsigned") || !strcmp(str, "void") || !strcmp(str, "static") ||
!strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "#include") ||
!strcmp(str, "main") || !strcmp(str, "printf"))
        return (true);
```

```
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
|| (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
&& str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Returns 'true' if the string is a COMMENT LINE.
bool isComment(char *str)
{
    strcpy(str, trim(str));
```

```
    int length = strlen(str);
    if (length < 2)
        return false;
    else if ((str[0] == str[1]) && (str[0] == '/'))
        return true;
    else if ((str[0] == str[length - 1]) && (str[1] == str[length - 2]))
    {
        if (str[0] == '/' && str[1] == '*')
            return true;
    }
    else
        return false;
}

// Returns 'true' if the string is a HEADER FILE.
bool isHeader(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);

    if ((str[length - 1] == 'h') && (str[length - 2] == '.'))
    {
        return true;
    }

    return false;
}

// Returns 'true' if the string contains a special charecter.
bool isSpecial(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str), f = 0;

    for (int i = 0; i < length; i++)
    {
        char ch = str[i];
        if (ch == '_')
            f = 0;

        else if (((int)ch >= 32 && (int)ch <= 47) || ((int)ch >= 58 && (int)ch <= 64) ||
((int)ch >= 91 && (int)ch <= 96) || ((int)ch >= 123 && (int)ch <= 127))
        {
            f = 1;
            break;
        }
        else
```

```
        {
            f = 0;
        }
    }

    if (f == 1)
        return true;

    return false;
}

// Returns true if the string is a format specifier
bool isFormat(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);

    if ((length <= 5) && (str[1] == '%'))
        return true;

    return false;
}

// Returns true if the specific keywwords have a spelling mistake in them
bool isSpellError(char *str)
{
    if (strcmp(str, "if") && strcmp(str, "for") && strcmp(str, "while") && strcmp(str,
"scanf") && strcmp(str, "printf") && strcmp(str, "gets") && strcmp(str, "getch") &&
strcmp(str, "main"))
    {
        return true;
    }

    return false;
}

// Extracts the SUBSTRING.
char *subString(char *str, int left, int right)
{
    strcpy(str, trim(str));
    int i;
    char *subStr = (char *)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
```

```
    return (subStr);
}

// Parsing the input STRING.
void parse(char *str)
{
    int left = 0, right = 0, flag;
    int len = strlen(str);
    char tempStr[30];

    FILE *fcode, *fspell;

    fcode = fopen("codefile.txt", "a");

    fspell = fopen("spellCheck.txt", "a");

    if (isComment(str))
    {
        fprintf(fcode, "'%s' IS A COMMENT LINE.\n", str);
    }
    else
    {
        while (right <= len && left <= right)
        {
            if (isDelimiter(str[right]) == false)
                right++;

            if (isDelimiter(str[right]) == true && left == right)
            {
                if (isOperator(str[right]) == true)
                    fprintf(fcode, "'%c' IS AN OPERATOR.\n", str[right]);

                right++;
                left = right;
            }
            else if (isDelimiter(str[right]) == true && left != right || (right == len &&
left != right))
            {
                char *subStr = subString(str, left, right - 1);

                char ch = str[right];

                if (strcmp(subStr, ""))
                {
                    if (ch == '{')
                    {
                        if (strcmp(subStr, "do") && strcmp(subStr, "else"))
```

```
                    {
                        fprintf(fspell, "'%s' has spelling error in it.\n", subStr);
                    }
                }

                if (ch == '(')
                {
                    if (isSpellError(subStr))
                    {
                        fprintf(fspell, "'%s' has spelling error in it.\n", subStr);
                    }
                }

                if (isKeyword(subStr) == true)
                {
                    fprintf(fcode, "'%s' IS A KEYWORD.\n", subStr);

                    strcpy(tempStr, subStr);
                }

                else if (isHeader(subStr) == true)
                    fprintf(fcode, "'%s' IS A HEADER FILE.\n", subStr);

                else if (isInteger(subStr) == true)
                    fprintf(fcode, "'%s' IS AN INTEGER (CONSTANT).\n", subStr);

                else if (isRealNumber(subStr) == true)
                    fprintf(fcode, "'%s' IS A REAL NUMBER.\n", subStr);

                else if (isFormat(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                    fprintf(fcode, "'%s' IS A FORMAT SPECIFIER.\n", subStr);

                else if (isSpecial(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                    fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT
CONTAINS A SPECIAL CHARACTER.\n", subStr);

                else if (validIdentifier(subStr) == true && isDelimiter(str[right -
1]) == false)
                {
                    fprintf(fcode, "'%s' IS A VALID IDENTIFIER.\n", subStr);

                    insert(subStr, tempStr);
                }
```

```
                    else if (validIdentifier(subStr) == false && isDelimiter(str[right -
1]) == false)
                        fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT EITHER
STARTS WITH A NUMBER.\n", subStr);

                    left = right;
                }
            }
        }
    }

    fclose(fcode);

    fclose(fspell);

    return;
}

// DRIVER FUNCTION
int main()
{
    char s[5000];
    int temp, cnt = 0;
    FILE *fptr, *fspellPtr;
    char str, strTwo, strThree;

    printf("================================================================================");
    printf("\n\tOn Completion of code, press (Ctrl+Z) followed by (Enter)\n");
    printf("================================================================================");

    printf("\nEnter your code here : \n");

    while (1)
    {
        temp = scanf("%[^\n]%*c", s);

        if (temp == -1)
        {
            break;
        }
        else
        {
            parse(s);

            cnt++;
        }
    }
```

```
    fptr = fopen("codefile.txt", "r");
    printf("\n\nThe lexical analysis of the code is : \n\n");
    str = fgetc(fptr);
    while (str != EOF)
    {
        printf("%c", str);
        str = fgetc(fptr);
    }
    printf("\n");
    fclose(fptr);

    printf("Total number of lines in program : %d\n\n", cnt);

    printf("\nFinal Symbol table : \n\n");

    display();

    printf("\n");

    printf("\nError Logs : \n\n");

    errorLogsRead = fopen("errorLogs.txt", "r");
    strTwo = fgetc(errorLogsRead);
    while (strTwo != EOF)
    {
        printf("%c", strTwo);
        strTwo = fgetc(errorLogsRead);
    }
    printf("\n\n");

    fclose(errorLogsRead);

    printf("Spelling errors in input code : \n\n");

    fspellPtr = fopen("spellCheck.txt", "r");
    strThree = fgetc(fspellPtr);
    while (strThree != EOF)
    {
        printf("%c", strThree);
        strThree = fgetc(fspellPtr);
    }
    printf("\n\n");

    fclose(fspellPtr);

    remove("codefile.txt");
```

```
    remove("errorLogs.txt");
    remove("spellCheck.txt");

    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 5> cd "d:\3rd
ck } ; if ($?) { .\spellCheck }
================================================================================
        On Completion of code, press (Ctrl+Z) followed by (Enter)
================================================================================
Enter your code here :
#include <stdio.h>
// Program to add two numbers
void main()
{
char chr;
int a = 5;
int b = 10;
int fro = 6;
f;
fi(b>a)
{
printf("%d",b);
}
eles{
printf("%d",a);
}
fro(int i=1;i<5;i++)
{
prinft("%d",i);
}
int 8var = 9;
int var$c = 15;
int d = a + b;
printf("%d",d);
}
^Z
```

```
The lexical analysis of the code is :

'#include' IS A KEYWORD.
'<' IS AN OPERATOR.
'stdio.h' IS A HEADER FILE.
'>' IS AN OPERATOR.
'// Program to add two numbers' IS A COMMENT LINE.
'void' IS A KEYWORD.
'main' IS A KEYWORD.
'char' IS A KEYWORD.
'chr' IS A VALID IDENTIFIER.
'int' IS A KEYWORD.
'a' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'5' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'b' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'10' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'fro' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'6' IS AN INTEGER (CONSTANT).
'f' IS A VALID IDENTIFIER.
'fi' IS A VALID IDENTIFIER.
'b' IS A VALID IDENTIFIER.
'>' IS AN OPERATOR.
'a' IS A VALID IDENTIFIER.
'printf' IS A KEYWORD.
'"%d"' IS A FORMAT SPECIFIER.
'b' IS A VALID IDENTIFIER.
'eles' IS A VALID IDENTIFIER.
'printf' IS A KEYWORD.
'"%d"' IS A FORMAT SPECIFIER.
'a' IS A VALID IDENTIFIER.
'fro' IS A VALID IDENTIFIER.
'int' IS A KEYWORD.
'i' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'1' IS AN INTEGER (CONSTANT).
'i' IS A VALID IDENTIFIER.
'<' IS AN OPERATOR.
'5' IS AN INTEGER (CONSTANT).
'i' IS A VALID IDENTIFIER.
'+' IS AN OPERATOR.
'+' IS AN OPERATOR.
'prinft' IS A VALID IDENTIFIER.
'"%d"' IS A FORMAT SPECIFIER.
'i' IS A VALID IDENTIFIER.
'int' IS A KEYWORD.
'8var' IS NOT A VALID IDENTIFIER BECAUSE IT EITHER STARTS WITH A NUMBER.
'=' IS AN OPERATOR.
'9' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'var$c' IS NOT A VALID IDENTIFIER BECAUSE IT CONTAINS A SPECIAL CHARACTER.
'=' IS AN OPERATOR.
'15' IS AN INTEGER (CONSTANT).
'int' IS A KEYWORD.
'd' IS A VALID IDENTIFIER.
'=' IS AN OPERATOR.
'a' IS A VALID IDENTIFIER.
'+' IS AN OPERATOR.
'b' IS A VALID IDENTIFIER.
'printf' IS A KEYWORD.
'"%d"' IS A FORMAT SPECIFIER.
'd' IS A VALID IDENTIFIER.

Total number of lines in program : 25
```

Final Symbol table :

SRL NO.	DATA TYPE	LABEL	ADDRESS
1	char	chr	1000
2	int	a	1001
3	int	b	1002
4	int	fro	1003
5	int	i	1004
6	int	d	1005

Error Logs :

```
No data type is mentioned for the variable f.
No data type is mentioned for the variable fi.
The variable name b already exists, hence the variable is no more inserted into symbol table.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name b already exists, hence the variable is no more inserted into symbol table.
No data type is mentioned for the variable eles.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name fro already exists, hence the variable is no more inserted into symbol table.
The variable name i already exists, hence the variable is no more inserted into symbol table.
The variable name i already exists, hence the variable is no more inserted into symbol table.
No data type is mentioned for the variable prinft.
The variable name i already exists, hence the variable is no more inserted into symbol table.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name b already exists, hence the variable is no more inserted into symbol table.
The variable name d already exists, hence the variable is no more inserted into symbol table.
```

Spelling errors in input code :

```
'fi' has spelling error in it.
'eles' has spelling error in it.
'fro' has spelling error in it.
'prinft' has spelling error in it.
```

NAME OF THE EXPERIMENT:
Write a C program to check whether value assigned to a variable matches the type of the variable.

Algorithm:
Step-1: Start.
Step-2: Create a user specific data structure called “symbolTab” that is used to contain 2 string variables representing the data type and variable name respectively along with an integer variable for the address allocation of the variable.
Step-3: Then we take two variables “first” and “last” to specify the first and last indexes of the symbol table along with a global variable size to specify the number of variables allotted in the program.
Step-4: Allocate a variable address to specify the starting address to allocate variables with respect to particular data types.
Step-5: Create two file pointers one for writing and another for reading an error logs file that were observed during compilation of the code.
Step-6: Create a function *trim(char *s) that takes a string as input and trims the extra white spaces from start and end of the string.
Step-7: Create a search function, that will check whether a variable received from the parser is already present within the symbol table or not.
Step-8: Create a function isKeywordVar to check is the variable is of the mention data types being either “int”, “float”, “long”, etc.
Step-9: Create a function insert that will check if the variable name is already present in the symbol table or not. If present, then it will place an error message within the errorLogs file else will insert the variable name with its data type and address within the user defined data type symbolTab.
Step-10: Create a function typeMatch that checks if datatype of variable is the same as the datatype of value assigned to it. If type is matched then matched message gets stored in a text file “typeMatch.txt” else an error message gets stored in the text file.
Step-11: Create a function “display” that will display the entire symbol table from the first pointer till the last created after reading the entire code entered by the user.
Step-12: Create a function isDelimiter(char ch) that takes character input to mark delimiter i.e. end of statement.
Step-13: Create a function isOperator(char ch) that takes character input to check whether it is an operator or not.
Step-14: Create a function validIdentifier(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not start with a number.
Step-15: Create a function isKeyword(char *str) that takes string input to check whether it is a keyword or not.
Step-16: Create a function isInteger(char *str) that takes string input to check whether it is an integer or not.
Step 17: Create a function isRealNumber(char *str) that takes string input to check whether it is a floating point number or not.

Step-18: Create a function isCharacter(char *str) that takes string input to check whether it is a character or not.
Step-19: Create a function isString(char *str) that takes string input to check whether the input is a string i.e. of char* type or not.
Step-20: Create a function isComment(char *str) that takes string input and checks whether the string starts with (‘//’) or (‘/*’ and ends with ‘*/’).
Step-21: Create a function isHeader(char *str) that takes string input and checks whether the string ends with ‘.h’ making it a header file.
Step-22: Create a function isSpecial(char *str) that takes a string input to check whether it is a valid identifier or not bsed to criteria that it does not contain a special character.
Step-23: Create a function isFormat(char *str) that takes a string input to check whether it is a format specifier or not based on whether it is of the pattern “%d” or “%ld”, etc.
Step-24: Create a function isSpellError(char *str) that takes string input to check whether it is spelling mistake with respect to keywords like “if”, “for”, etc.
Step-25: Create a function *subString(char *str, int left, int right) that takes a string, and 2 indexes left and right as input to extract the substring form a larger string str.
Step-26: Create a function parse(char *str) that takes a large string as input. Then declare a file pointer *fcode. Then open a file temporary “codefile.txt” that is pointed by fcode.
Step-27: Then check with the boolear function isComment(str) if the string is a comment or not.
Step-28: If the string is not a comment line, the we start checking each string based on index extraction of strings from left and right, thereby checking isDelimiter(str[right]), then increment right by 1.
Step-29: if isDelimiter(str[right]) == true && left == right, then we check if isOperator(str[right]) == true, to store in the file that the character is an operator. If the operator is ‘=’ then set a flag variable fbr as 1 else set it at 0.
Step-30: else if isDelimiter(str[right]) == true && left ≠ right || (right == len && left ≠ right), then char *subStr = subString(str, left, right - 1)
Step-31: We check if delimiter character is a ‘{‘, then we check if the preceding string has spelling mistake with respect to words “do” or else.
Step-32: We check if the delimiter character is a ‘(‘, then we check if the preceding string has spelling mistake with respect to words “if” or “for” or “while” or “scanf” or “printf” or “gets” or “getch” or “main” via the function isSpellError(char *str).
Step-33: Then we check whether the substring is a keyword or header file or integer or real number or character or string or format specifier or contains special character or is a valid identifier based on the boolean functions we created above and save the corresponding message in the file. If an assignment case is observed, then type matching is initiated between the variable type and it’s assigned value with the typeMatch(char *str).
Step-34: If the variable is a keyword, we copy it into a temporary variable.

Step-35: If we find a valid identifier, then check if it is preceded by a data type specified earlier. If it is specified, then we pass the variable to our insert function else we place a message into the error log file. The valid variable name is stored in a temporary variable to be used in the type matching function.
Step-36: Then we make left = right and close the file finally with fclose(fcode).
Step-37: Within the main function, we ask the user to enter the code and terminate entering by pressing “Ctrl+Z” followed by “Enter”.
Step-38: We create another file pointer FILE *fptr.
Step-39: As the user enters code line by line, we use a counter to count the number of lines in the program input by the user.
Step-40: Then we print the lexical analysis of the input code by taking the first string from the file str = fgetc(fptr).
Step-41: While we do not reach the end of file, we print the content of the file.
Step-42: Then we close the file with fclose(fptr) and print the total number of lines in the program.
Step-43: We then print the entire symbol table with the display function mentioned above.
Step-44: We start reading the error logs file to print any error (if any) within the code we passed into our program and print them.
Step-45: We start reading the spellCheck file to print any spelling errors (if any) within the code we passed into our program and print them.
Step-46: We start reading the typeMatch file to print appropriate message for variables according to the value assigned to each of them based on the code we passed into the program.
Step-47: Finally, we remove both the temporary files with remove("codefile.txt"), remove("errorLogs.txt"), remove("spellCheck.txt") and remove(“typeMatch.txt”) respectively.
Step-48: STOP

Source code:

```
#include <stdbool.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define null 0

// Symbol table created
struct symbolTab
{
    char dataType[20];
    char varName[20];
    int address;
    struct symbolTab *next;
};
```

```
// First and last pointers of symbol table
struct symbolTab *first, *last;

// Initially the size is zero
int size = 0;

// Initialize the primary address to allocate the variables to memory
int addr = 1000;

// File to note the errors in the code
FILE *errorLogs, *errorLogsRead;

// Returns string after removing extra white spaces
char *trim(char *s)
{
    int i;

    while (isspace(*s))
        s++;
    for (i = strlen(s) - 1; (isspace(s[i])); i--)
        ;
    s[i + 1] = '\0';
    return s;
}

// Searches for a particular variable name to avoid duplication
int search(char var[])
{
    int i, flag = 0;
    struct symbolTab *p;
    p = first;
    for (i = 0; i < size; i++)
    {
        if (strcmp(p->varName, var) == 0)
        {
            flag = 1;
        }
        p = p->next;
    }
    return flag;
}

// Returns 'true' if the string is a Keyword representing a variable
bool isKeywordVar(char *str)
{
    if (!strcmp(str, "int") || !strcmp(str, "float") || !strcmp(str, "char") ||
!strcmp(str, "double") || !strcmp(str, "long"))
```

```
        return true;

    return false;
}

// Function to insert a variable into symbol table
void insert(char *str, char *datType)
{
    int n;
    n = search(str);

    errorLogs = fopen("errorLogs.txt", "a");

    if (n == 1)
    {
        fprintf(errorLogs, "The variable name %s already exists, hence the variable is no
more inserted into symbol table.\n", str);
    }
    else
    {
        if (isKeywordVar(datType))
        {
            struct symbolTab *p;
            p = malloc(sizeof(struct symbolTab));
            strcpy(p->dataType, datType);
            strcpy(p->varName, str);
            p->address = addr++;
            p->next = null;
            if (size == 0)
            {
                first = p;
                last = p;
            }
            else
            {
                last->next = p;
                last = p;
            }
            size++;
        }
        else
        {
            fprintf(errorLogs, "No data type is mentioned for the variable %s.\n", str);
        }
    }
    fclose(errorLogs);
}
```

```
// Function to check if datatype of variable matches with value assigned to it
void typeMatch(char *str, int dataCat)
{
    FILE *fmatch;
    fmatch = fopen("typeMatch.txt", "a");
    strcpy(str, trim(str));
    int i;
    struct symbolTab *p;
    p = first;
    for (i = 0; i < size; i++)
    {
        if (!strcmp(str, p->varName))
        {
            if ((!strcmp(p->dataType, "int") && dataCat == 0) || (!strcmp(p->dataType,
"float") && dataCat == 1) || (!strcmp(p->dataType, "char") && dataCat == 2) ||
(!strcmp(p->dataType, "char*") && dataCat == 3))
            {
                fprintf(fmatch, "Data type matches assigned value for variable %s, hence
No Error.\n", str);
            }
            else
            {
                fprintf(fmatch, "Type mismatch error for variable %s.\n", str);
            }

            break;
        }
        p = p->next;
    }

    fclose(fmatch);
}

// Displays the final symbol table
void display()
{
    int i;
    struct symbolTab *p;
    p = first;
    printf("SRL NO.\t\tDATA TYPE\tLABEL\t\tADDRESS\n");
    for (i = 0; i < size; i++)
    {
        printf("%d\t\t%s\t\t%s\t\t%d\n", i + 1, p->dataType, p->varName, p->address);
        p = p->next;
    }
}
```

```
// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char *str)
{
    strcpy(str, trim(str));
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);

    return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char *str)
{
    strcpy(str, trim(str));
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") ||
!strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str,
"case") || !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") ||
!strcmp(str, "unsigned") || !strcmp(str, "void") || !strcmp(str, "static") ||
```

```
!strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "#include") ||
!strcmp(str, "main") || !strcmp(str, "printf"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
|| (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char *str)
{
    strcpy(str, trim(str));
    int i, len = strlen(str);
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] !=
'4' && str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9'
&& str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Returns 'true' if the string is a character
```

```
bool isCharacter(char *str)
{
    strcpy(str, trim(str));
    int len = strlen(str);

    if (str[0] == '\'' && str[len - 1] == '\'')
        return true;

    return false;
}

// Returns 'true' if the string is of string type
bool isString(char *str)
{
    strcpy(str, trim(str));
    int len = strlen(str);

    if (str[0] == '\"' && str[len - 1] == '\"')
        return true;

    return false;
}

// Returns 'true' if the string is a COMMENT LINE.
bool isComment(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);
    if (length < 2)
        return false;
    else if ((str[0] == str[1]) && (str[0] == '/'))
        return true;
    else if ((str[0] == str[length - 1]) && (str[1] == str[length - 2]))
    {
        if (str[0] == '/' && str[1] == '*')
            return true;
    }
    else
        return false;
}

// Returns 'true' if the string is a HEADER FILE.
bool isHeader(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);
```

```
    if ((str[length - 1] == 'h') && (str[length - 2] == '.'))
    {
        return true;
    }

    return false;
}

// Returns 'true' if the string contains a special charecter.
bool isSpecial(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str), f = 0;

    for (int i = 0; i < length; i++)
    {
        char ch = str[i];
        if (ch == '_')
            f = 0;

        else if (((int)ch >= 32 && (int)ch <= 47) || ((int)ch >= 58 && (int)ch <= 64) ||
((int)ch >= 91 && (int)ch <= 96) || ((int)ch >= 123 && (int)ch <= 127))
        {
            f = 1;
            break;
        }
        else
        {
            f = 0;
        }
    }

    if (f == 1)
        return true;

    return false;
}

// Returns true if the string is a format specifier
bool isFormat(char *str)
{
    strcpy(str, trim(str));
    int length = strlen(str);

    if ((length <= 5) && (str[1] == '%'))
        return true;
```

```
    return false;
}

// Returns true if the specific keywwords have a spelling mistake in them
bool isSpellError(char *str)
{
    if (strcmp(str, "if") && strcmp(str, "for") && strcmp(str, "while") && strcmp(str,
"scanf") && strcmp(str, "printf") && strcmp(str, "gets") && strcmp(str, "getch") &&
strcmp(str, "main"))
    {
        return true;
    }

    return false;
}

// Extracts the SUBSTRING.
char *subString(char *str, int left, int right)
{
    strcpy(str, trim(str));
    int i;
    char *subStr = (char *)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char *str)
{
    int left = 0, right = 0, flag, fbr = 0, setCat = -1;
    int len = strlen(str);
    char tempStr[30], tempVar[30];

    FILE *fcode, *fspell;

    fcode = fopen("codefile.txt", "a");

    fspell = fopen("spellCheck.txt", "a");

    if (isComment(str))
    {
        fprintf(fcode, "'%s' IS A COMMENT LINE.\n", str);
    }
```

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

            if (isDelimiter(str[right]) == true && left == right)
            {
                if (isOperator(str[right]) == true)
                {
                    fprintf(fcode, "'%c' IS AN OPERATOR.\n", str[right]);

                    if (str[right] == '=')
                    {
                        fbr = 1;
                    }
                    else
                    {
                        fbr = 0;
                    }
                }

                right++;
                left = right;
            }
            else if (isDelimiter(str[right]) == true && left != right || (right == len &&
left != right))
            {
                char *subStr = subString(str, left, right - 1);

                char ch = str[right];

                if (strcmp(subStr, ""))
                {
                    if (ch == '{')
                    {
                        if (strcmp(subStr, "do") && strcmp(subStr, "else"))
                        {
                            fprintf(fspell, "'%s' has spelling error in it.\n", subStr);
                        }
                    }

                    if (ch == '(')
                    {
                        if (isSpellError(subStr))
                        {
```

```
            fprintf(fspell, "'%s' has spelling error in it.\n", subStr);
        }
    }

    if (isKeyword(subStr) == true)
    {
        fprintf(fcode, "'%s' IS A KEYWORD.\n", subStr);

        strcpy(tempStr, subStr);
    }

    else if (isHeader(subStr) == true)
        fprintf(fcode, "'%s' IS A HEADER FILE.\n", subStr);

    else if (isInteger(subStr) == true)
    {
        fprintf(fcode, "'%s' IS AN INTEGER (CONSTANT).\n", subStr);
        setCat = 0;

        if (fbr != 0)
        {
            typeMatch(tempVar, setCat);
        }
    }

    else if (isRealNumber(subStr) == true)
    {
        fprintf(fcode, "'%s' IS A FLOATING POINT NUMBER.\n", subStr);
        setCat = 1;

        if (fbr != 0)
        {
            typeMatch(tempVar, setCat);
        }
    }

    else if (isCharacter(subStr) == true)
    {
        fprintf(fcode, "'%s' IS A CHARACTER.\n", subStr);
        setCat = 2;

        if (fbr != 0)
        {
            typeMatch(tempVar, setCat);
        }
    }
```

```
                    else if (isString(subStr) == true)
                    {
                        fprintf(fcode, "'%s' IS A STRING.\n", subStr);
                        setCat = 3;

                        if (fbr != 0)
                        {
                            typeMatch(tempVar, setCat);
                        }
                    }

                    else if (isFormat(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                        fprintf(fcode, "'%s' IS A FORMAT SPECIFIER.\n", subStr);

                    else if (isSpecial(subStr) == true && isDelimiter(str[right - 1]) ==
false)
                        fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT
CONTAINS A SPECIAL CHARACTER.\n", subStr);

                    else if (validIdentifier(subStr) == true && isDelimiter(str[right -
1]) == false)
                    {
                        fprintf(fcode, "'%s' IS A VALID IDENTIFIER.\n", subStr);

                        insert(subStr, tempStr);

                        strcpy(tempVar, subStr);
                    }

                    else if (validIdentifier(subStr) == false && isDelimiter(str[right -
1]) == false)
                        fprintf(fcode, "'%s' IS NOT A VALID IDENTIFIER BECAUSE IT EITHER
STARTS WITH A NUMBER.\n", subStr);

                    left = right;
                }
            }
        }
    }

    fclose(fcode);

    fclose(fspell);

    return;
}
```

```
// DRIVER FUNCTION
int main()
{
    char s[5000];
    int temp, cnt = 0;
    FILE *fptr, *fspellPtr, *fmatchPtr;
    char str, strTwo, strThree, strFour;

    printf("================================================================================");
    printf("\n\tOn Completion of code, press (Ctrl+Z) followed by (Enter)\n");
    printf("================================================================================");

    printf("\nEnter your code here : \n");

    while (1)
    {
        temp = scanf("%[^\n]%*c", s);

        if (temp == -1)
        {
            break;
        }
        else
        {
            parse(s);

            cnt++;
        }
    }

    fptr = fopen("codefile.txt", "r");
    printf("\n\nThe lexical analysis of the code is : \n\n");
    str = fgetc(fptr);
    while (str != EOF)
    {
        printf("%c", str);
        str = fgetc(fptr);
    }
    printf("\n");
    fclose(fptr);

    printf("Total number of lines in program : %d\n\n", cnt);

    printf("\nFinal Symbol table : \n\n");

    display();
```

```
printf("\n");

printf("\nError Logs : \n\n");

errorLogsRead = fopen("errorLogs.txt", "r");
strTwo = fgetc(errorLogsRead);
while (strTwo != EOF)
{
    printf("%c", strTwo);
    strTwo = fgetc(errorLogsRead);
}
printf("\n\n");

fclose(errorLogsRead);

printf("Spelling errors in input code : \n\n");

fspellPtr = fopen("spellCheck.txt", "r");
strThree = fgetc(fspellPtr);
while (strThree != EOF)
{
    printf("%c", strThree);
    strThree = fgetc(fspellPtr);
}
printf("\n\n");

fclose(fspellPtr);

printf("Type matching errors in input code : \n\n");

fmatchPtr = fopen("typeMatch.txt", "r");
strFour = fgetc(fmatchPtr);
while (strFour != EOF)
{
    printf("%c", strFour);
    strFour = fgetc(fmatchPtr);
}
printf("\n\n");

fclose(fmatchPtr);

remove("codefile.txt");
remove("errorLogs.txt");
remove("spellCheck.txt");
remove("typeMatch.txt");
```

```
    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 6\Matching Types>
{ gcc typeMismatch.c -o typeMismatch } ; if ($?) { .\typeMismatch }
================================================================================
        On Completion of code, press (Ctrl+Z) followed by (Enter)
================================================================================
Enter your code here :
#include <stdio.h>
// Program to add two numbers
void main()
{
char chr;
int a = 5;
int b = 10;
int fro = 6;
int br = 6.6;
char cc = 20.8;
f;
fi(b>a)
{
printf("%d",b);
}
eles{
printf("%d",a);
}
fro(int i=1;i<5;i++)
{
prinft("%d",i);
}
int 8var = 9;
int var$c = 15;
int d = a + b;
printf("%d",d);
}
^Z
```

```
Total number of lines in program : 27


Final Symbol table :

SRL NO.         DATA TYPE       LABEL           ADDRESS
1               char            chr             1000
2               int             a               1001
3               int             b               1002
4               int             fro             1003
5               int             br              1004
6               char            cc              1005
7               int             i               1006
8               int             d               1007


Error Logs :

No data type is mentioned for the variable f.
No data type is mentioned for the variable fi.
The variable name b already exists, hence the variable is no more inserted into symbol table.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name b already exists, hence the variable is no more inserted into symbol table.
No data type is mentioned for the variable eles.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name fro already exists, hence the variable is no more inserted into symbol table.
The variable name i already exists, hence the variable is no more inserted into symbol table.
The variable name i already exists, hence the variable is no more inserted into symbol table.
No data type is mentioned for the variable prinft.
The variable name i already exists, hence the variable is no more inserted into symbol table.
The variable name a already exists, hence the variable is no more inserted into symbol table.
The variable name b already exists, hence the variable is no more inserted into symbol table.
The variable name d already exists, hence the variable is no more inserted into symbol table.

            Spelling errors in input code :

            'fi' has spelling error in it.
            'eles' has spelling error in it.
            'fro' has spelling error in it.
            'prinft' has spelling error in it.


            Type matching errors in input code :

            Data type matches assigned value for variable a, hence No Error.
            Data type matches assigned value for variable b, hence No Error.
            Data type matches assigned value for variable fro, hence No Error.
            Type mismatch error for variable br.
            Type mismatch error for variable cc.
            Data type matches assigned value for variable i, hence No Error.
```

NAME OF THE EXPERIMENT:
Write a program to find the first of the following grammar:
S- ABC
A - a/b/ephsilon
B - c/d/ephsilon
C - e/f/ ephsilon

Algorithm:
Step-1: Start.
Step-2: Declare global variables like count, k, e, n = 0 under integer category and variables calc_first[10][100], production[10][10], first[10], ck under character category.
Step-3: Create a function findfirst(char c, int q1, int q2) and check if the character is a terminal, then store it within the array first[]. Then we check the production matrix to check if the character is in the first place or in the last place. If the string gets terminated, then the results are stored within first, else if the query checking is not for the character in the first place, then a recursive call is made to the function findfirst(char c, int q1, int q2), else the check for terminal epsilon, which in this case is '#' is checked.
Step-4: Within the main method define few variables like jm, km, choice, i, count. Then the production matrix is set for the grammar we want to derive.
Step-5: Start a nested loop with outer limit till count for all rows of production and inner limit till 100 to assign the matrix calc_first[k][kay] = '!'.
Step-6: For ach row, check if the First of c ha already been calculated or not, where c = production[k][0].
Step-7: If it is checked we continue to next variable else we call the function findfirst(c, 0, 0); and increment ptr by 1.
Step-8: Then we add c to the calculated list with done[ptr] = c;
Step-9: We fill the array calc_first[point1][point2++] = c;
Step-10: Then the first for the grammar is printed for each variable present within the grammar based on the terminals.
Step-11: STOP

Source code:
```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>

int count, n = 0, k, e;
char calc_first[10][100], production[10][10], first[10], ck;

void findfirst(char c, int q1, int q2)
{
    int j;

    if (!(isupper(c)))
    {
```

```
        first[n++] = c;
    }
    for (j = 0; j < count; j++)
    {
        if (production[j][0] == c)
        {
            if (production[j][2] == '#')
            {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

int main()
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 10;

    strcpy(production[0], "S-ABC");
    strcpy(production[1], "A-a");
    strcpy(production[2], "A-b");
    strcpy(production[3], "A-#");
    strcpy(production[4], "B-c");
    strcpy(production[5], "B-d");
    strcpy(production[6], "B-#");
    strcpy(production[7], "C-e");
    strcpy(production[8], "C-f");
    strcpy(production[9], "C-#");

    int kay;
    char done[count];
    int ptr = -1;

    for (k = 0; k < count; k++)
    {
```

```
        for (kay = 0; kay < 100; kay++)
        {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;

    printf("\n First for input Grammar : ");

    for (k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;

        if (xxx == 1)
            continue;

        findfirst(c, 0, 0);
        ptr += 1;

        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        for (i = 0 + jm; i < n; i++)
        {
            int lark = 0, chk = 0;

            for (lark = 0; lark < point2; lark++)
            {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
```

```
    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\
 Grammar\" ; if ($?) { gcc FirstGrammar.c -o FirstGra

 First for input Grammar :
 First(S) = { a, b, c, d, e, f, #, }

 First(A) = { a, b, #, }

 First(B) = { c, d, #, }
```

NAME OF THE EXPERIMENT:
Write a program to find the following of the below grammar:
S - AaAb / BbBa
A - ephsilon
B – ephsilon

Algorithm:
Step-1: Start.
Step-2: Declare function follow(char c) to calculate follow.
Step-3: Declare global variables like count, k, e, n = 0 under integer category and variables calc_first[10][100], production[10][10], first[10], ck under character category.
Step-4: Create a function findfirst(char c, int q1, int q2) and check if the character is a terminal, then store it within the array first[]. Then we check the production matrix to check if the character is in the first place or in the last place. If the string gets terminated, then the results are stored within first, else if the query checking is not for the character in the first place, then a recursive call is made to the function findfirst(char c, int q1, int q2), else the check for terminal epsilon, which in this case is '#' is checked.
Step-5: Create a function followfirst(char c, int c1, int c2) and check if a terminal is encountered at the particular character. Then start a loop based on the count of production rules. Within the loop check if calc_first[i][0] == c, then break. Then we include the first set of the non-terminal in the follow of the original array. In case we reach the end of profuction, then we call the follow(production[c1][0]), else we call the recursive function followfirst(production[c1][c2], c1, c2+1).
Step-6: Create function follow(char c), to add "$" to the follow set of the start symbol. Then calculate the first of the next non-terminal in the production by running the recursive function followfirst(production[i][j+1], i, (j+2)). If if(production[i][j+1]=='\0' && c≠production[i][0]), then calculate the non-terminal in the L.H.S of the production.
Step-7: Within the main method define few variables like jm, km, choice, i, count. Then the production matrix is set for the grammar we want to derive.
Step-8: Start a nested loop with outer limit till count for all rows of production and inner limit till 100 to assign the matrix calc_first[k][kay] = '!'.
Step-9: For ach row, check if the First of c ha already been calculated or not, where c = production[k][0].
Step-10: If it is checked we continue to next variable else we call the function findfirst(c, 0, 0); and increment ptr by 1.
Step-11: Then we add c to the calculated list with done[ptr] = c;
Step-12: We fill the array calc_first[point1][point2++] = c;
Step-13: Initialize the calc_follow array based on the count of the production rues.
Step-14: Then check for each production rule check if ck == donee[kay]. If it is already done, the continue else call the follow(ck) and increment ptr by 1.
Step-15: Then add ck to the calculated list donee[ptr] = ck;

Step-16: Then print the follow set of the grammar by storing calc_follow[point1][point2++] = f[i]; to the calc_follow matrix.
Step-17: STOP

Source code:

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

void follow(char c);

int count, n = 0, k, e, m = 0;
char calc_first[10][100], production[10][10], first[10], f[10], ck, calc_follow[10][100];

void findfirst(char c, int q1, int q2)
{
    int j;

    if (!(isupper(c)))
    {
        first[n++] = c;
    }
    for (j = 0; j < count; j++)
    {
        if (production[j][0] == c)
        {
            if (production[j][2] == '#')
            {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
```

```
    if (!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for (i = 0; i < count; i++)
        {
            if (calc_first[i][0] == c)
                break;
        }

        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}

void follow(char c)
{
    int i, j;

    if (production[0][0] == c)
    {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 2; j < 10; j++)
        {
            if (production[i][j] == c)
            {
                if (production[i][j + 1] != '\0')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }

                if (production[i][j + 1] == '\0' && c != production[i][0])
```

```
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

int main()
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 4;

    strcpy(production[0], "S-AaAb");
    strcpy(production[1], "S-BbBa");
    strcpy(production[2], "A-#");
    strcpy(production[3], "B-#");

    int kay;
    char done[count];
    int ptr = -1;

    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;

    printf("\n Follow for input Grammar : ");

    for (k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        xxx = 0;

        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;

        if (xxx == 1)
            continue;

        findfirst(c, 0, 0);
        ptr += 1;

        done[ptr] = c;
```

```
        calc_first[point1][point2++] = c;

        for (i = 0 + jm; i < n; i++)
        {
            int lark = 0, chk = 0;

            for (lark = 0; lark < point2; lark++)
            {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                calc_first[point1][point2++] = first[i];
            }
        }

        jm = n;
        point1++;
    }

    char donee[count];
    ptr = -1;

    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for (e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;

        follow(ck);
        ptr += 1;
```

```
        donee[ptr] = ck;
        printf("\n Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        for (i = 0 + km; i < m; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }

    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler
($?) { gcc FollowGrammar.c -o FollowGrammar }

Follow for input Grammar :
Follow(S) = { $,  }


Follow(A) = { a, b,  }


Follow(B) = { b, a,  }
```

NAME OF THE EXPERIMENT:
Create a parse tree for the following language W ⟶ id * id + id
Given grammar is S ⟶ T + S | T
T ⟶ id * T | id | (S)

Algorithm:
Step-1: Start.
Step-2: Declare a function isUpper (char c) that checks if the input character passed as parameter is a character or not.
Step-3: Declare a function substring (char *str, int left, int right) that extracts the substring from the input string based on the indexes of left and right
Step-4: In the main method, declare the production matrix, the input string, count and reference string.
Step-5: Enter the production rules of grammar in the production matrix in a synchronised order using strcpy() function.
Step-6: Print the production rules of the original grammar from the production matrix created above.
Step-7: Take the language input from the user and store it inside the string s
Step-8: Print the updates occurring in the original grammar starting with the original form from production [0].
Step-9: Start a loop for the number of rows of the production rules according to the production matrix.
Step-10: Retrieve the right part of the original production with the substring function.
Step-10: The take a temporary string modStr where the string and characters are concatenated.
Step-11: For each production rules with values in common, corresponding string or character is applied based on the condition within another nested loop.
Step-12: Then check whether the final temporary string generated has uppercase characters in it or not at each step.
Step-13: If the final string has no upper-case letters and the output matches the input language, then use a flag variable.
Step-14: If the flag variable is 1 then print the particular language can be retrieved from the given grammar.
Step-15: Else print the particular language cannot be retrieved from the given grammar.
Step-16: STOP

Source code:
```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

int isUpper(char c)
{
```

```
    if (c >= 'A' && c <= 'Z')
        return 1;
    else
        return 0;
}

char *subString(char *str, int left, int right)
{
    int i;
    char *subStr = (char *)malloc(
        sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

int main()
{
    char production[10][10], s[200];
    char *refStr;
    int cnt = 5, f = -1;

    strcpy(production[0], "S-T+S");
    strcpy(production[1], "T-c*T");
    strcpy(production[2], "S-T");
    strcpy(production[3], "T-c");
    strcpy(production[4], "T-(S)");

    printf("\nOriginal Grammar: \n");

    for (int i = 0; i < cnt; i++)
    {
        printf("%s", production[i]);
        printf("\n");
    }

    printf("\nEnter Language : ");
    scanf("%[^\n]%*c", s);

    refStr = subString(s, 2, strlen(s));

    printf("\n\nUpdates in main Production Grammar : \n");
    printf("%s", production[0]);

    for (int i = 1; i < cnt; i++)
    {
        char *shortStr = subString(production[0], 2, strlen(production[0]));
        char modStr[30] = "";

        strcat(modStr, "S-");

        for (int k = 0; k < strlen(shortStr); k++)
```

```
        {
            if (shortStr[k] == production[i][0])
            {
                char *useStr = subString(production[i], 2, strlen(production[i]));
                strcat(modStr, useStr);
            }
            else
            {
                strncat(modStr, &shortStr[k], 1);
            }
        }

        printf("\n%s", modStr);
        strcpy(production[0], modStr);

        char *finStr = subString(modStr, 2, strlen(modStr));

        int terminate = 0;
        for (int r = 0; r < strlen(finStr); r++)
        {
            if (isUpper(finStr[r]))
            {
                terminate = 1;
                break;
            }
        }

        if (!strcmp(finStr, refStr) && terminate != 1)
        {
            f = 1;
            break;
        }
    }

    if (f == 1)
    {
        printf("\n\n%s language can be derived from the Original grammar.\n\n", s);
    }
    else
    {
        printf("\n\n%s language cannot be derived from the Original grammar.\n\n", s);
    }

    return 0;
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day
\" ; if ($?) { gcc ParseTree.c -o ParseTree } ; if ($?) { .\

Original Grammar:
S-T+S
T-c*T
S-T
T-c
T-(S)

Enter Language : W-c*c+c


Updates in main Production Grammar :
S-T+S
S-c*T+S
S-c*T+T
S-c*c+c

W-c*c+c language can be derived from the Original grammar.
```

NAME OF THE EXPERIMENT:
Write a C program to check whether the given grammar will be accepted by LL (1) parser.
S⟶aSbS | bSaS | ephsilon

Algorithm:
Step-1: Start.
Step-2: Declare a matrix table of int type, 2 arrays terminal and nonterminal of character type.
Step-3: Declare a structure product having a string and an integer in it.
Step-4: Declare number of productions, first and follow matrix along with another first_rhs matrix.
Step-5: Create a function that checks if a symbol is a non-terminal or not.
Step-6: Declare a function readFile that reads the input from the inputFile.txt.
Step-7: Based on the terminal and nonterminal, the production rules are generated on the string buffer.
Step-8: Declare a function add_FIRST_A_to_FOLLOW_B(char A, char B) to work on the first method for production rules.
Step-9: Declare another function add_FOLLOW_A_to_FOLLOW_B(char A, char B) to add the follow method for production rules.
Step-10: Declare function Follow that generates the follow terminals of the input grammar.
Step-11: Declare function add_FIRST_A_to_FIRST_B(char A, char B) to add elements to first of production rules based on need.
Step-12: Declare the First function that generates the first terminals of the input grammar.
Step-13: Declare function add_FIRST_A_to_FIRST_RHS__B(char A, int B) to work for the first rhs method based on production rules.
Step-14: Decare function FIRST_RHS() to generate the first from the right hand side of the production rules.
Step-15: In the main method, print the input grammar first followed by the first and the follows terminals var each variable present in the production rules.
Step-16: The rules for checking are defined considering ‘#’ as the epsilon.
Step-17: Check if any of the derived strings for parse table contains null ending due to multiple production rules satisfying the same condition of each variable.
Step-18: In case of null string print, the particular grammar is not accepted by LL (1) parser.
Step-19: In case of no null string print, the particular grammar is accepted by LL (1) parse and a parse table can be generated from it.
Step-20: STOP

Source code:
```
// Here '#' represents epsilon
#include <stdio.h>
#include <string.h>
#define TSIZE 128

int table[100][TSIZE];
```

```
char terminal[TSIZE];
char nonterminal[26];

struct product
{
    char str[100];
    int len;
} pro[20];

// no of productions in form A->ß
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];

// stores first of each production in form A->ß
char first_rhs[100][TSIZE];

// check if the symbol is nonterminal
int isNT(char c)
{
    return c >= 'A' && c <= 'Z';
}

// reading data from the file
void readFromFile()
{
    FILE *fptr;
    fptr = fopen("inputFile.txt", "r");
    char buffer[255];
    int i;
    int j;
    while (fgets(buffer, sizeof(buffer), fptr))
    {
        printf("%s", buffer);
        j = 0;
        nonterminal[buffer[0] - 'A'] = 1;
        for (i = 0; i < strlen(buffer) - 1; ++i)
        {
            if (buffer[i] == '|')
            {
                ++no_pro;
                pro[no_pro - 1].str[j] = '\0';
                pro[no_pro - 1].len = j;
                pro[no_pro].str[0] = pro[no_pro - 1].str[0];
                pro[no_pro].str[1] = pro[no_pro - 1].str[1];
                pro[no_pro].str[2] = pro[no_pro - 1].str[2];
                j = 3;
            }
            else
            {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>')
                {
```

```
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}

void add_FIRST_A_to_FOLLOW_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}

void add_FOLLOW_A_to_FOLLOW_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}

void FOLLOW()
{
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro)
    {
        for (k = 0; k < 26; ++k)
        {
            if (!nonterminal[k])
                continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i)
            {
                for (j = 3; j < pro[i].len; ++j)
                {
                    if (nt == pro[i].str[j])
                    {
                        for (x = j + 1; x < pro[i].len; ++x)
                        {
                            char sc = pro[i].str[x];
                            if (isNT(sc))
                            {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['#'])
                                    continue;
```

```
                            }
                            else
                            {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                            add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
                    }
                }
            }
        }
    }
}

void add_FIRST_A_to_FIRST_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
        {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}

void FIRST()
{
    int i, j;
    int t = 0;
    while (t < no_pro)
    {
        for (i = 0; i < no_pro; ++i)
        {
            for (j = 3; j < pro[i].len; ++j)
            {
                char sc = pro[i].str[j];
                if (isNT(sc))
                {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['#'])
                        continue;
                }
                else
                {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['#'] = 1;
        }
```

```
        ++t;
    }
}

void add_FIRST_A_to_FIRST_RHS__B(char A, int B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}

// Calculates FIRST(ß) for each A->ß
void FIRST_RHS()
{
    int i, j;
    int t = 0;
    while (t < no_pro)
    {
        for (i = 0; i < no_pro; ++i)
        {
            for (j = 3; j < pro[i].len; ++j)
            {
                char sc = pro[i].str[j];
                if (isNT(sc))
                {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['#'])
                        continue;
                }
                else
                {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first_rhs[i]['#'] = 1;
        }
        ++t;
    }
}

int main()
{
    printf("\nInput Grammar: \n");
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k, f = -1;
```

```
    // display first of each variable
    printf("\n\n");
    for (i = 0; i < no_pro; ++i)
    {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
        {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j)
            {
                if (first[c - 'A'][j])
                {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }

    // display follow of each variable
    printf("\n");
    for (i = 0; i < no_pro; ++i)
    {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
        {
            char c = pro[i].str[0];
            printf("FOLLOW OF %c: ", c);
            for (j = 0; j < TSIZE; ++j)
            {
                if (follow[c - 'A'][j])
                {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }
    // display first of each variable ß
    // in form A->ß
    printf("\n");
    for (i = 0; i < no_pro; ++i)
    {
        printf("FIRST OF %s: ", pro[i].str);
        for (j = 0; j < TSIZE; ++j)
        {
            if (first_rhs[i][j])
            {
                printf("%c ", j);
            }
        }
        printf("\n");
    }
```

```
terminal['$'] = 1;
terminal['#'] = 0;

// printing parse table
printf("\n");

int p = 0;
for (i = 0; i < no_pro; ++i)
{
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
        p = p + 1;
    for (j = 0; j < TSIZE; ++j)
    {
        if (first_rhs[i][j] && j != '#')
        {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['#'])
        {
            for (k = 0; k < TSIZE; ++k)
            {
                if (follow[pro[i].str[0] - 'A'][k])
                {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
k = 0;
char *cpStr;
for (i = 0; i < no_pro; ++i)
{
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
    {
        // printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j)
        {
            if (table[k][j])
            {
                cpStr = pro[table[k][j] - 1].str;
                if (cpStr[strlen(cpStr) - 1] == '>')
                {
                    f = 1;
                    break;
                }
            }
        }
    }
}

if (f == 1)
{
    printf("Grammar is not accepted by LL(1) parser");
```

```
    }
    else
    {
        printf("Grammar is accepted by LL(1) parser");
    }

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

Output:

```
PS D:\3rd Year 6th Sem All Materials\Com
r LL1 accepted\" ; if ($?) { gcc LL1Pars

Input Grammar:
S->aSbS|bSaS|#

FIRST OF S: # a b

FOLLOW OF S: $ a b

FIRST OF S->aSbS: a
FIRST OF S->bSaS: b
FIRST OF S->: #

Grammar is not accepted by LL(1) parser
```

NAME OF THE EXPERIMENT:
Create the LL (1) parse table of the following grammar.
S ⟶(L) | a
L ⟶ SL'
L' ⟶ ephsilon | ,SL'

Algorithm:
Step-1: Start.
Step-2: Declare a matrix table of int type, 2 arrays terminal and nonterminal of character type.
Step-3: Declare a structure product having a string and an integer in it.
Step-4: Declare number of productions, first and follow matrix along with another first_rhs matrix.
Step-5: Create a function that checks if a symbol is a non-terminal or not.
Step-6: Declare a function readFile that reads the input from the inputFile.txt.
Step-7: Based on the terminal and nonterminal, the production rules are generated on the string buffer.
Step-8: Declare a function add_FIRST_A_to_FOLLOW_B(char A, char B) to work on the first method for production rules.
Step-9: Declare another function add_FOLLOW_A_to_FOLLOW_B(char A, char B) to add the follow method for production rules.
Step-10: Declare function Follow that generates the follow terminals of the input grammar.
Step-11: Declare function add_FIRST_A_to_FIRST_B(char A, char B) to add elements to first of production rules based on need.
Step-12: Declare the First function that generates the first terminals of the input grammar.
Step-13: Declare function add_FIRST_A_to_FIRST_RHS__B(char A, int B) to work for the first rhs method based on production rules.
Step-14: Decare function FIRST_RHS() to generate the first from the right hand side of the production rules.
Step-15: In the main method, print the input grammar first followed by the first and the follows terminals var each variable present in the production rules.
Step-16: The rules for checking are defined considering ‘#’ as the epsilon.
Step-17: Finally print the parse table generated within the table matrix via the production structure indexing of the string and its particular length. This gives the final parse table via 2 nested loops.
Step-18: STOP

Source code:
```
// Here '#' represents epsilon
#include <stdio.h>
#include <string.h>
#define TSIZE 128

int table[100][TSIZE];
char terminal[TSIZE];
char nonterminal[26];
```

```
struct product
{
    char str[100];
    int len;
} pro[20];

// no of productions in form A->ß
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];

// stores first of each production in form A->ß
char first_rhs[100][TSIZE];

// check if the symbol is nonterminal
int isNT(char c)
{
    return c >= 'A' && c <= 'Z';
}

// reading data from the file
void readFromFile()
{
    FILE *fptr;
    fptr = fopen("inputFile.txt", "r");
    char buffer[255];
    int i;
    int j;
    while (fgets(buffer, sizeof(buffer), fptr))
    {
        printf("%s", buffer);
        j = 0;
        nonterminal[buffer[0] - 'A'] = 1;
        for (i = 0; i < strlen(buffer) - 1; ++i)
        {
            if (buffer[i] == '|')
            {
                ++no_pro;
                pro[no_pro - 1].str[j] = '\0';
                pro[no_pro - 1].len = j;
                pro[no_pro].str[0] = pro[no_pro - 1].str[0];
                pro[no_pro].str[1] = pro[no_pro - 1].str[1];
                pro[no_pro].str[2] = pro[no_pro - 1].str[2];
                j = 3;
            }
            else
            {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>')
                {
                    terminal[buffer[i]] = 1;
                }
```

```
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}

void add_FIRST_A_to_FOLLOW_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}

void add_FOLLOW_A_to_FOLLOW_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}

void FOLLOW()
{
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro)
    {
        for (k = 0; k < 26; ++k)
        {
            if (!nonterminal[k])
                continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i)
            {
                for (j = 3; j < pro[i].len; ++j)
                {
                    if (nt == pro[i].str[j])
                    {
                        for (x = j + 1; x < pro[i].len; ++x)
                        {
                            char sc = pro[i].str[x];
                            if (isNT(sc))
                            {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['#'])
                                    continue;
                            }
                            else
```

```
                            {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                            add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
                    }
                }
            }
        }
    }
}

void add_FIRST_A_to_FIRST_B(char A, char B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
        {
            first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
        }
    }
}

void FIRST()
{
    int i, j;
    int t = 0;
    while (t < no_pro)
    {
        for (i = 0; i < no_pro; ++i)
        {
            for (j = 3; j < pro[i].len; ++j)
            {
                char sc = pro[i].str[j];
                if (isNT(sc))
                {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['#'])
                        continue;
                }
                else
                {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['#'] = 1;
        }
        ++t;
    }
```

```
}

void add_FIRST_A_to_FIRST_RHS__B(char A, int B)
{
    int i;
    for (i = 0; i < TSIZE; ++i)
    {
        if (i != '#')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}

// Calculates FIRST(ß) for each A->ß
void FIRST_RHS()
{
    int i, j;
    int t = 0;
    while (t < no_pro)
    {
        for (i = 0; i < no_pro; ++i)
        {
            for (j = 3; j < pro[i].len; ++j)
            {
                char sc = pro[i].str[j];
                if (isNT(sc))
                {
                    add_FIRST_A_to_FIRST_RHS__B(sc, i);
                    if (first[sc - 'A']['#'])
                        continue;
                }
                else
                {
                    first_rhs[i][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first_rhs[i]['#'] = 1;
        }
        ++t;
    }
}

int main()
{
    printf("\nInput Grammar: \n");
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k;

    // display first of each variable
```

```
    printf("\n\n");
    for (i = 0; i < no_pro; ++i)
    {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
        {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j)
            {
                if (first[c - 'A'][j])
                {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }

    // display follow of each variable
    printf("\n");
    for (i = 0; i < no_pro; ++i)
    {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
        {
            char c = pro[i].str[0];
            printf("FOLLOW OF %c: ", c);
            for (j = 0; j < TSIZE; ++j)
            {
                if (follow[c - 'A'][j])
                {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }
    // display first of each variable ß
    // in form A->ß
    printf("\n");
    for (i = 0; i < no_pro; ++i)
    {
        printf("FIRST OF %s: ", pro[i].str);
        for (j = 0; j < TSIZE; ++j)
        {
            if (first_rhs[i][j])
            {
                printf("%c ", j);
            }
        }
        printf("\n");
    }

    terminal['$'] = 1;
    terminal['#'] = 0;
```

```
// printing parse table
printf("\n");
printf("\n\t**************** LL(1) PARSING TABLE ******************\n");
printf("\t-----------------------------------------------------------\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i)
{
    if (terminal[i])
        printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i)
{
    if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
        p = p + 1;
    for (j = 0; j < TSIZE; ++j)
    {
        if (first_rhs[i][j] && j != '#')
        {
            table[p][j] = i + 1;
        }
        else if (first_rhs[i]['#'])
        {
            for (k = 0; k < TSIZE; ++k)
            {
                if (follow[pro[i].str[0] - 'A'][k])
                {
                    table[p][k] = i + 1;
                }
            }
        }
    }
}
k = 0;
for (i = 0; i < no_pro; ++i)
{
    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0]))
    {
        printf("%-10c", pro[i].str[0]);
        for (j = 0; j < TSIZE; ++j)
        {
            if (table[k][j])
            {
                printf("%-10s", pro[table[k][j] - 1].str);
            }
            else if (terminal[j])
            {
                printf("%-10s", "");
            }
        }
        ++k;
        printf("\n");
```

```
        }
    }

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

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 7\Creat
 Table\" ; if ($?) { gcc ParseTable.c -o ParseTable } ; if ($?) { .'

Input Grammar:
S->(L)|a
L->SR
R->#|,SR

FIRST OF S: ( a
FIRST OF L: ( a
FIRST OF R: # ,

FOLLOW OF S:   $ ,
FOLLOW OF L: )
FOLLOW OF R:

FIRST OF S->(L): (
FIRST OF S->a: a
FIRST OF L->SR : ( a
FIRST OF R->#: #
FIRST OF R->,S: ,


        ********** LL(1) PARSING TABLE **********
        ------------------------------------------------
                  $        (        )        ,        a
S                          S->(L)                     S->a
L                          L->SR                      L->SR
R         R->#                                R->,S
```

NAME OF THE EXPERIMENT:
Implement LL(1) parser with stack to show that it accepts the given grammar
S⟶ AA
A⟶aA
A⟶b

Algorithm:
Step-1: Start.
Step-2: Declare functions followfirst(char, int, int), findfirst(char, int, int), and follow(char c) along with global variables count, n = 0, calc_first[10][100], calc_follow[10][100], m = 0, production[10][10], first[10], f[10], k, ck, and e.
Step-3: Declare the main, where user has to first input the number of productions.
Step-4: Then user has to enter the production rules in the format like A-B, where A and B are grammar symbols.
Step-5: Then save each of the productions in the production matrix.
Step-6: Then start calculating the first and follow of each variables and store them in matrixes calc_first[][] and calc_follow[][] respectively.
Step-7: Within the function check if the particular variable or terminal has been calculated before or not. If calculated before then continue else find the first and follow and store them in the respective matrices.
Step-8: Within the loop check if !isupper(production[k][kay]) && production[k][kay] ≠ '#' && production[k][kay] ≠ '=' && production[k][kay] ≠ '\0', and if true, we we check the production matrix with respect to the terminal array ter[ap].
Step-9: Then print the first and follow from the calc_first[][] and calc_follow[][] matrices we calculated above.
Step-10: Then print the LL (1) parse table with the terminals as columns and the variables as rows.
Step-11: For each terminal association with a variable, we print the production rule associated from the production matrix we created before.
Step-12: Then change the production pointer table based on the values of the production rules.
Step-13: Finally, after printing the entire parse table by taking help from the associated functions, discussed above, the user is asked to enter the string for which the grammar checking is supposed to occur.
Step-14: Then take a stack of char type and size of 100 characters.
Step-15: Enter the first production rule as input for the stack entering ‘$’ and ‘S’ respectively according to the example we have used.
Step-16: Then based on the top pointer of the stack, enter the value assigned to a specific variable within the stack, and pop that variable from the stack.
Step-17: When a terminal is encountered at the top of the stack, it is matched with characters from the input string with the help of the look ahead header.
Step-18: If the character value is found to be a match, then pop the terminal from the stack as well as from the input string.
Step-19: Continue steps 16 to 18 until the stack becomes empty after popping all the characters.

Step-20: The entire operation discussed above is represented in form of a table with the “Stack”, “Input” and “Action” columns present in it.
Step-21: If for a terminal, the pop operation cannot be performed due to no character matching between the top of stack and look ahead header, then break out from the loop giving error result that input string is not accepted by LL (1) parser.
Step-22: If all the characters of the input string are not checked even after traversing throughout the stack, then an error message is displayed that this language is not accepted by LL (1) parser.
Step-23: If above scenarios do not occur, then the message is displayed that the language is accepted by LL (1) parser, with the help of the stack.
Step-24: STOP

Source code:

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    printf("\nNumber of productions : ");
    scanf("%d", &count);
    printf("\nEnter %d productions as A-B (A and B are grammar symbols) : \n", count);
    for (i = 0; i < count; i++)
    {
        scanf("%s%c", production[i], &ch);
    }
    int kay;
    char done[count];
    int ptr = -1;
    for (k = 0; k < count; k++)
    {
```

```
        for (kay = 0; kay < 100; kay++)
        {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;
    for (k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        findfirst(c, 0, 0);
        ptr += 1;
        done[ptr] = c;
        printf("\nFirst(%c)= { ", c);
        calc_first[point1][point2++] = c;
        for (i = 0 + jm; i < n; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("-----------------------------------------------\n\n");
    char donee[count];
    ptr = -1;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
```

```
    int land = 0;
    for (e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        land += 1;
        follow(ck);
        ptr += 1;
        donee[ptr] = ck;
        printf("Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;
        for (i = 0 + km; i < m; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
    char ter[10];
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] != '#' &&
production[k][kay] != '-' && production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
```

```
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer is :-");

printf("\n\t\t\t=======================================================================
=======================================\n");
    printf("\t\t\t\t|\t");
    for (ap = 0; ap < sid; ap++)
    {
        printf("%c\t\t", ter[ap]);
    }

printf("\n\t\t\t=======================================================================
=======================================\n");
    char first_prod[count][sid];
    for (ap = 0; ap < count; ap++)
    {
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while (production[ap][k] != '\0')
        {
            if (!isupper(production[ap][k]))
            {
                tem[ct++] = production[ap][k];
                tem[ct++] = '_';
                tem[ct++] = '\0';
                k++;
                break;
            }
            else
            {
                int zap = 0;
                int tuna = 0;
                for (zap = 0; zap < count; zap++)
                {
                    if (calc_first[zap][0] == production[ap][k])
                    {
                        for (tuna = 1; tuna < 100; tuna++)
```

```
                    {
                        if (calc_first[zap][tuna] != '!')
                        {
                            tem[ct++] = calc_first[zap][tuna];
                        }
                        else
                            break;
                    }
                    break;
                }
            }
            tem[ct++] = '_';
        }
        k++;
    }
    int zap = 0, tuna;
    for (tuna = 0; tuna < ct; tuna++)
    {
        if (tem[tuna] == '#')
        {
            zap = 1;
        }
        else if (tem[tuna] == '_')
        {
            if (zap == 1)
            {
                zap = 0;
            }
            else
                break;
        }
        else
        {
            first_prod[ap][destiny++] = tem[tuna];
        }
    }
}
char table[land][sid + 1];
ptr = -1;
for (ap = 0; ap < land; ap++)
{
    for (kay = 0; kay < (sid + 1); kay++)
    {
        table[ap][kay] = '!';
    }
}
for (ap = 0; ap < count; ap++)
{
    ck = production[ap][0];
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (ck == table[kay][0])
            xxx = 1;
    if (xxx == 1)
```

```
            continue;
        else
        {
            ptr = ptr + 1;
            table[ptr][0] = ck;
        }
    }
    for (ap = 0; ap < count; ap++)
    {
        int tuna = 0;
        while (first_prod[ap][tuna] != '\0')
        {
            int to, ni = 0;
            for (to = 0; to < sid; to++)
            {
                if (first_prod[ap][tuna] == ter[to])
                {
                    ni = 1;
                }
            }
            if (ni == 1)
            {
                char xz = production[ap][0];
                int cz = 0;
                while (table[cz][0] != xz)
                {
                    cz = cz + 1;
                }
                int vz = 0;
                while (ter[vz] != first_prod[ap][tuna])
                {
                    vz = vz + 1;
                }
                table[cz][vz + 1] = (char)(ap + 65);
            }
            tuna++;
        }
    }
    for (k = 0; k < sid; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            if (calc_first[k][kay] == '!')
            {
                break;
            }
            else if (calc_first[k][kay] == '#')
            {
                int fz = 1;
                while (calc_follow[k][fz] != '!')
                {
                    char xz = production[k][0];
                    int cz = 0;
                    while (table[cz][0] != xz)
```

```
                    {
                        cz = cz + 1;
                    }
                    int vz = 0;
                    while (ter[vz] != calc_follow[k][fz])
                    {
                        vz = vz + 1;
                    }
                    table[k][vz + 1] = '#';
                    fz++;
                }
                break;
            }
        }
    }
    for (ap = 0; ap < land; ap++)
    {
        printf("\t\t\t   %c\t|\t", table[ap][0]);
        for (kay = 1; kay < (sid + 1); kay++)
        {
            if (table[ap][kay] == '!')
                printf("\t\t");
            else if (table[ap][kay] == '#')
                printf("%c=#\t\t", table[ap][0]);
            else
            {
                int mum = (int)(table[ap][kay]);
                mum -= 65;
                printf("%s\t\t", production[mum]);
            }
        }
        printf("\n");

printf("\t\t\t-------------------------------------------------------------------------
-----------------------------------");
        printf("\n");
    }
    int j;
    printf("\n\nEnter STRING for Grammar Checking : ");
    char input[100];
    scanf("%s%c", input, &ch);

printf("\n\t\t\t\t\t==========================================================================
======\n");
    printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

printf("\n\t\t\t\t\t==========================================================================
======\n");
    int i_ptr = 0, s_ptr = 1;
    char stack[100];
    stack[0] = '$';
    stack[1] = table[0][0];
    while (s_ptr != -1)
    {
```

```
printf("\t\t\t\t\t\t");
int vamp = 0;
for (vamp = 0; vamp <= s_ptr; vamp++)
{
    printf("%c", stack[vamp]);
}
printf("\t\t\t");
vamp = i_ptr;
while (input[vamp] != '\0')
{
    printf("%c", input[vamp]);
    vamp++;
}
printf("\t\t\t");
char her = input[i_ptr];
char him = stack[s_ptr];
s_ptr--;
if (!isupper(him))
{
    if (her == him)
    {
        i_ptr++;
        printf("POP ACTION\n");
    }
    else
    {
        printf("\nString Not Accepted by LL(1) Parser !!\n");
        exit(0);
    }
}
else
{
    for (i = 0; i < sid; i++)
    {
        if (ter[i] == her)
            break;
    }
    char produ[100];
    for (j = 0; j < land; j++)
    {
        if (him == table[j][0])
        {
            if (table[j][i + 1] == '#')
            {
                printf("%c=#\n", table[j][0]);
                produ[0] = '#';
                produ[1] = '\0';
            }
            else if (table[j][i + 1] != '!')
            {
                int mum = (int)(table[j][i + 1]);
                mum -= 65;
                strcpy(produ, production[mum]);
                printf("%s\n", produ);
```

```
                    }
                    else
                    {
                        printf("\nString Not Accepted by LL(1) Parser !!\n");
                        exit(0);
                    }
                }
            }
            int le = strlen(produ);
            le = le - 1;
            if (le == 0)
            {
                continue;
            }
            for (j = le; j >= 2; j--)
            {
                s_ptr++;
                stack[s_ptr] = produ[j];
            }
        }
    }

printf("\n\t\t\t=======================================================================
=======================================\n");
    if (input[i_ptr] == '\0')
    {
        printf("\t\t\t\t\t\t\t\tINPUT STRING IS ACCEPTED BY LL(1) PARSER \n");
    }
    else
        printf("\n\t\t\t\t\t\t\t\tINPUT STRING REJECTED BY LL(1) PARSER \n");

printf("\t\t\t=======================================================================
=======================================\n\n");
}

void follow(char c)
{
    int i, j;
    if (production[0][0] == c)
    {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 2; j < 10; j++)
        {
            if (production[i][j] == c)
            {
                if (production[i][j + 1] != '\0')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                {
```

```
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    if (!(isupper(c)))
    {
        first[n++] = c;
    }
    for (j = 0; j < count; j++)
    {
        if (production[j][0] == c)
        {
            if (production[j][2] == '#')
            {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
    if (!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for (i = 0; i < count; i++)
        {
            if (calc_first[i][0] == c)
                break;
        }
```

```
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Output:

```
PS D:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 8> cd "d:\3rd Year 6th Sem All Materials\Compiler Design\Lab\Day 8\" ; if ($?)
ck.c -o LL1ParseTableUsingStack } ; if ($?) { .\LL1ParseTableUsingStack }

Number of productions : 3

Enter 3 productions as A-B (A and B are grammar symbols) :
S-AA
A-aA
A-b

First(S)= { a, b, }

First(A)= { a, b, }

--------------------------------------------

Follow(S) = { $,  }

Follow(A) = { a, b, $,  }

                                The LL(1) Parsing Table for the above grammer is :-
================================================================================================================
       |      -              a              b              $
================================================================================================================
  S    |                     S-AA           S-AA
----------------------------------------------------------------------------------------------------------------
  A    |                     A-aA           A-b
----------------------------------------------------------------------------------------------------------------

Enter STRING for Grammar Checking : abab$
```

Stack	Input	Action
$S	abab$	S-AA
$AA	abab$	A-aA
$AAa	abab$	POP ACTION
$AA	bab$	A-b
$Ab	bab$	POP ACTION
$A	ab$	A-aA
$Aa	ab$	POP ACTION
$A	b$	A-b
$b	b$	POP ACTION
$	$	POP ACTION

INPUT STRING IS ACCEPTED BY LL(1) PARSER

Enter STRING for Grammar Checking : abab

Stack	Input	Action
$S	abab	S-AA
$AA	abab	A-aA
$AAa	abab	POP ACTION
$AA	bab	A-b
$Ab	bab	POP ACTION
$A	ab	A-aA
$Aa	ab	POP ACTION
$A	b	A-b
$b	b	POP ACTION
$		

String Not Accepted by LL(1) Parser !!