INDEX

Name Rohit Anil Burewar Enrolment No./PRN No. 230400294

Branch : Artificial Intelligence Year / Semester : 4th yr / 7th sem

Subject : ~~Deep Learning~~ CD Roll No. 66.

Exp. No.	Name of Experiment	Page No.	Date	Grade	Sign
01.	Write a lex program to count the number of words, characters, blank spaces, & lines in a given input text.	01	15/7	A1	23/7/2024
02.	Write a Lex program to identify identifiers and special characters.	02	23/7	A1	30/7/2024
03.	Write a lex program to find whether a given no. is even or odd.	03	30/7	A1	5/8/2024
04.	Write a lex program to check whether a no. is prime or not	04	5/8	A1	13/8/2024
05.	Write a lex program to find the addition of two numbers.	05	13/8	A1	20/8/2024
06.	Write a lex program to calculate first() of a given CFG.	06	20/8	A1	27/8/2024

Exp. No.	Name of Experiment	Page No.	Date	Grade
07.	Write a lex program to calculate follow() in give CFG.	07	31/9	A+
08.	Write a program to check given CFG are LL(1) or not.	08	7/9	A+
09.	Write a lex program to check whether given code LR(0) or not.	09	24/9	A+
10.	Write a program to generate three addrss code.	10	08/10	A+

Date :

Practical NO. 1

* **Aim** :- Write a program to demonsteate the use of count number of words, characters, blank spaces, and liner in a given input text.

* **Theory** :-

Lex is a lexical analyzer generator that is used in compiler desiang to identify lexical tokens from input text. It works on the principle of regular expression & generates a C program which performs pattern matching on the input.

- Lexical Analyzer :

It reads source code character by character & group them into meaningful sequence called lexems. These are matched against patterns to produce tokens which are used by parser.

- Lex is used to count the number of characters, words, blank spaces, & liner from input text.

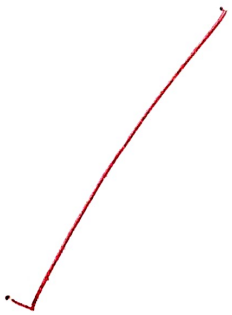
* **Result** :- Hence a program executed successfully.

Page No.

Practical1

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

int word_count = 0;
int char_count = 0;
int space_count = 0;
int line_count = 0;
%}

%%
[ \t]+       { space_count += yyleng; char_count += yyleng; }
\n           { line_count++; char_count++; }
[^\t\n ]+    { word_count++; char_count += yyleng; }
%%

int yywrap() {
   return 1;
}

int main() {
   printf("Enter text (Ctrl+D to end on Linux/Mac, Ctrl+Z on Windows):\n");
   yylex();
   printf("\nNumber of words: %d\n", word_count);
   printf("Number of characters: %d\n", char_count);
   printf("Number of blank spaces: %d\n", space_count);
   printf("Number of lines: %d\n", line_count);
   return 0;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\prc1.l.txt"

C:\Users\shahu>gcc lex.yy.c -o prc1.exe

C:\Users\shahu>prc1.exe
Enter text (Ctrl+D to end on Linux/Mac, Ctrl+Z on Windows):
i am 21 years
my email id : shahu@gmail.com
^Z

Number of words: 9
Number of characters: 46
Number of blank spaces: 9
Number of lines: 2
```



Date :

Practical No. 02

* Aim :- Write a Lex program to identify identifiers & special character.

* Theory :-

Lex is a lexical analyzer generator used in compiler design to recognise tokens from input text.

- Identifiers are names given to variables, functions, etc. usually starting with a letter or underscore and followed by letter, digit, or underscores.

- Special characters are symbols like +, -, *, /, =, ;, ,, (,), {, }, ; etc. which have predefined meaning in programming languages.

* Result :- Hence a program executed successfully.

Practical 2

```
%{
#include <stdio.h>
%}
/* Regular definitions */
IDENT   [a-zA-Z_][a-zA-Z0-9_]*
SPECIAL [!@#$%^&*(){};:,.<>/?|+=\-]

%%
{IDENT}     { printf("Identifier: %s\n", yytext); }
{SPECIAL}   { printf("Special character: %s\n", yytext); }
[ \t\n]+    ;   /* ignore whitespace */
.           { printf("Other: %s\n", yytext); }
%%

int yywrap(void) { return 1; }
int main() {
    printf("Enter input (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac):\n");
    yylex();
    return 0;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\pr2.l.txt"

C:\Users\shahu>gcc lex.yy.c -o pr2.exe

C:\Users\shahu>pr2.exe
Enter input (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac):
hello my 4 brothers
Identifier: hello
Identifier: my
Other: 4
Identifier: brothers
^Z

C:\Users\shahu>
```

Date :

Practical No. 3

* **Aim :-** Write a lex program to find whether a given no. is even or odd.

* **Theory :-**

Lex is a tool used in compiler design to generate lexical analyzer. It match input against patterns using regular expression & performs action written in C.

- In this program, lex is used to recognize number from the input. A number is classified as :

* Even if its last digit is 0, 2, 4, 6 or 8.

* Odd if its last digit is 1, 3, 5, 7 or 9.

* **Result :-** Hence a program executed successfully.

Practical 3

```
%{
#include <stdio.h>
#include <stdlib.h>   // for atoi
%}

%%
[0-9]+   {
          int num = atoi(yytext);
          if (num % 2 == 0)
             printf("%d is Even\n", num);
          else
             printf("%d is Odd\n", num);
       }
[ \t\n]+ ;   /* ignore whitespace */
.       { printf("Invalid input: %s\n", yytext); }
%%

int yywrap(void) { return 1; }
int main() {
   printf("Enter a number (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac):\n");
   yylex();
   return 0;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\pr3.l.txt"

C:\Users\shahu>gcc lex.yy.c -o pr3.exe

C:\Users\shahu>pr3.exe
Enter a number (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac):
45
45 is Odd
67
67 is Odd
48
48 is Even
52
52 is Even
```

Date :

Practical NO. 4

* Aim :- Write a lex program to check whether a no. is prime or not.

* Theory :-

Lex is a lexical analyzer generator used in compiler design. It identifies tokens using regular expressions & executes action written in C.

In this program, lex reads an input number & checks if it is prime. A prime number is a number greater than 1 & itself.

* If the number is divisible by any value bet^ 2 & n/2, it is not prime.
* otherwise, it is prime.

* Results: Hence a program executed successfully.

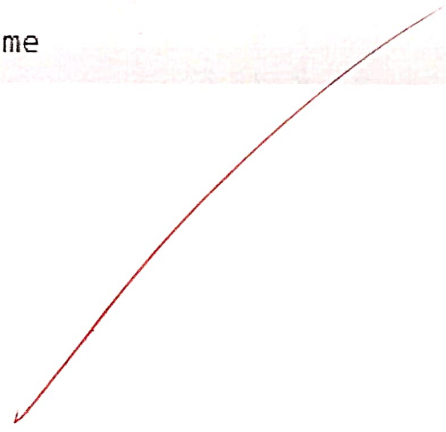
Page No.

Practical 4

```
%{
#include <stdio.h>
#include <stdlib.h>   // for atoi

/* Function to check prime */
int isPrime(int n) {
   if (n <= 1) return 0;
   if (n == 2) return 1;
   if (n % 2 == 0) return 0;
   for (int i = 3; i*i <= n; i += 2) {
      if (n % i == 0) return 0;
   }
   return 1;
}
%}

%%
[0-9]+   {
          int num = atoi(yytext);
          if (isPrime(num))
             printf("%d is Prime\n", num);
          else
             printf("%d is Not Prime\n", num);
        }
 [ \t\n]+  ;   /* ignore whitespace */
 .       { printf("Invalid input: %s\n", yytext); }
 %%

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

```
int main() {
    printf("Enter numbers (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac to stop):\n");
    yylex();
    return 0;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\pr4.l.txt"

C:\Users\shahu>gcc lex.yy.c -o pr4.exe

C:\Users\shahu>pr4.exe
Enter numbers (Ctrl+Z then Enter on Windows, Ctrl+D on Linux/Mac to stop):
3
3 is Prime
45
45 is Not Prime
23
23 is Prime
157
157 is Prime
177
177 is Not Prime
```



Date :

Practical NO.5

* Aim :- Write a Lex program to find the addition of two numbers.

* Theory :-

Lex is a tool used in compiler design to generate lexical analyser. It uses regular expressions to recognize token & perform action in c code.

- In program, lex identifies two number from input, convert them to integers, & perform addition. This demonstrates how lex can handle numeric token recognition & arithmetic operations.

* Result :- Hence a program executed successfully.

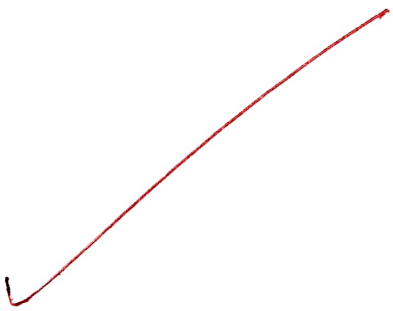
Page No.

Practical 5

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

int num1 = -1, num2 = -1;   // store the two numbers
%}

%%
[0-9]+   {
            if (num1 == -1)
               num1 = atoi(yytext);
            else
               num2 = atoi(yytext);
         }
[ \t\n]+  ;   /* ignore whitespace */
.        { printf("Invalid input: %s\n", yytext); }
%%

int yywrap(void) { return 1; }

int main() {
   printf("Enter two numbers: ");
   yylex();

   if (num1 != -1 && num2 != -1)
      printf("Sum = %d\n", num1 + num2);
   else
      printf("Please enter two valid numbers.\n");
```

```
    return 0;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\pr5.1.txt"

C:\Users\shahu>gcc lex.yy.c -o pr5.exe

C:\Users\shahu>pr5.exe
Enter two numbers: 34 127
^Z
Sum = 161
```



Date :

Practical No. 06

* Aim :- Write a lex program to calculate the first() of a given context free grammer.

* Theory :-

In compiler design, first() is a function used in syntax analysis (parsing). for a given grammer's symbol x, first(x) is the set of terminals that appear as first symbols of strings derived from x.

* If $x \rightarrow a$, where a is a terminal then a ∈ first(x).

* If $x \rightarrow Y...$, where Y is a non-terminal, then first(Y) ⊆ first(x).

* If a production can derive ε, then ε ∈ first(x).

* Result :- Hence a program executed succesfully.

Page No.

Practical 6

```
%{
#include <stdio.h>
#include <string.h>
#define MAX 20
#define MAX_RHS 10

typedef struct {
   char lhs[20];
   char rhs[MAX_RHS][20];
   int rhs_len;
} Production;
Production prods[MAX];
int pcount = 0;
char first[MAX][MAX][20];    // first[nt][symbol]
int first_count[MAX];
int nt_count = 0;
char nts[MAX][20];          // non-terminals
int is_nonterminal(char *sym) {
   for(int i=0; i<nt_count; i++)
      if(strcmp(nts[i], sym) == 0)
         return 1;
   return 0;
}

int nt_index(char *sym) {
   for(int i=0; i<nt_count; i++)
      if(strcmp(nts[i], sym) == 0)
         return i;
   return -1;
}

void add_first(int nt, char *sym) {
   for(int i=0; i<first_count[nt]; i++) {
      if(strcmp(first[nt][i], sym) == 0)
         return;
   }
   strcpy(first[nt][first_count[nt]++], sym);
}

// Compute FIRST sets with epsilon propagation
void compute_first() {
   int changed;
```

```
                    if(!is_nonterminal(lhs))
                        strcpy(nts[nt_count++], lhs);
                    } else {
                        strcpy(rhs[rhs_len++], yytext);
                    }
                }
"->"            { rhs_len = 0; }
\n              {
                  if(lhs_ready) {
                    strcpy(prods[pcount].lhs, lhs);
                    prods[pcount].rhs_len = rhs_len;

                    for(int i=0; i<rhs_len; i++)
                        strcpy(prods[pcount].rhs[i], rhs[i]);
                    pcount++;
                    lhs_ready = 0;
                    rhs_len = 0;
                  }
                }
.               ; // ignore any other char

%%

int main() {
    printf("Enter CFG productions line by line (no | symbol). Ctrl+D to end:\n");

    while(yylex()) {}
    compute_first();
    print_first();

    return 0;
}
```

```
(base) computer@computer:~/Music$ flex cfg1.l
(base) computer@computer:~/Music$ gcc lex.yy.c -o cfg1 -lfl
(base) computer@computer:~/Music$ ./cfg1
Enter CFG productions line by line (no | symbol). Ctrl+D to end:
S ->a A B
S ->b A
S ->ε
A ->a A b
A ->ε
B ->b B
B ->c
FIRST(S) = { a, b, ε }
FIRST(A) = { a, ε }
FIRST(B) = { b, c }
```

Date :

Practical No. 7

* Aim :- Write a lex Program to calculate follow () in given context free grammar.

* Theory :-

In compiler design, follow() is a function used in syntax analysis, while first () helps determine which terminal symbol can appear at the start of a derivation. follow() tells us which terminal symbols can appear immediately to right of a non-terminal in same derivation.

- start symbol
  If S is the start symbol, then $ (end of input marker) ∈ follow (S).

* Result :- Hence a program executed successfully.

Practical 7

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 20

typedef struct {
   char lhs[10];
   char rhs[10][10];
   int rhs_len;
} Production;
Production prods[100];
int pcount = 0;
char nts[MAX][10];            // Non-terminals
int nt_count = 0;
char first[MAX][MAX][10];     // FIRST sets
char follow[MAX][MAX][10];    // FOLLOW sets
int first_cnt[MAX], follow_cnt[MAX];
char lhs[10], rhs[10][10];    // Temporary buffers
int rhs_len = 0, lhs_ready = 0;
int is_nt(char *sym) { return isupper(sym[0]); }

int nt_idx(char *sym) {
   for (int i = 0; i < nt_count; i++)
      if (!strcmp(nts[i], sym)) return i;
   return -1;
}
void add_nt(char *sym) {
   if (nt_idx(sym) == -1)
      strcpy(nts[nt_count++], sym);
}
void add_sym(char arr[MAX][10], int *count, char *sym) {
   for (int i = 0; i < *count; i++)
      if (!strcmp(arr[i], sym)) return;
   strcpy(arr[(*count)++], sym);
}
void add_first(int i, char *sym)  { add_sym(first[i], &first_cnt[i], sym); }
void add_follow(int i, char *sym) { add_sym(follow[i], &follow_cnt[i], sym); }

void compute_first() {
   int changed;
   do {
```

```
                strcpy(prods[pcount].lhs, lhs);
                prods[pcount].rhs_len = rhs_len;
                for (int i = 0; i < rhs_len; i++)
                    strcpy(prods[pcount].rhs[i], rhs[i]);
                pcount++;
                lhs_ready = 0;
                rhs_len = 0;
            }
        }
    .       ;

%%

int main() {
    printf("Enter productions (one per line), e.g.:\nS -> A B\nUse ε for epsilon. Press Ctrl+D to finish.\n");
    while (yylex()) {}
    compute_first();
    compute_follow();
    print_follow();
    return 0;
}
```

```
(base) computer@computer:~/Music$ flex fllcfg.l
(base) computer@computer:~/Music$ gcc lex.yy.c -o fllcfg -lfl
(base) computer@computer:~/Music$ ./fllcfg
Enter productions (e.g., A -> a B). Use ε for epsilon. Ctrl+D to finish.
S ->a A B
S ->b A
S ->ε
A ->a A b
A ->ε
B ->b B
B ->c
FOLLOW(S) = { $ }
FOLLOW(A) = { b, c, $ }
FOLLOW(B) = { $ }
```

Date :

Practical No. 08

* Aim :- Write a program to check given context free grammar (CFG) are LL(1) or not.

* Theory :-

In compiler design, parsing is the process of analysing whether a sequence of tokens follows the grammer of a programing language predictive parser like LL(1) parser on first() & follow() set to decide which production rule to apply without backtracking.

- A grammer is called LL(1) if it can be parsed by an LL(1) parser.

- This ensures that parser can always decide which production to use by looking at the next input symbol without confusion.

* Result : Hence a program executed successfully.

Practical 8

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 20

char productions[MAX][50];
int prod_count = 0;
int check_left_recursion();
int check_left_factoring();
%}

%%
[A-Z]("->")[a-zA-Z|()+*]+  {
   strcpy(productions[prod_count++], yytext);
}
\n   ;
.    ;
%%

int check_left_recursion() {
   for(int i=0;i<prod_count;i++) {
      char nt = productions[i][0];   // LHS non-terminal
      char rhs[50];
      strcpy(rhs, productions[i]+3); // RHS after "->"
      char *token = strtok(rhs,"|");
      while(token) {
         if(token[0]==nt) {
            printf(" Left recursion detected in: %s\n", productions[i]);
            return 1;
         }
         token = strtok(NULL,"|");
      }
   }
   return 0;
}

int check_left_factoring() {
   for(int i=0;i<prod_count;i++) {
      char rhs[50];
      strcpy(rhs, productions[i]+3);
      char *alt1 = strtok(rhs,"|");
      while(alt1) {
         char *alt2 = strtok(NULL,"|");
```

```
            if(alt2 && alt1[0]==alt2[0]) {
                printf(" Left factoring needed in: %s\n", productions[i]);
                return 1;
            }
            alt1 = alt2;
        }
    }
    return 0;
}

int main() {
    printf("Enter grammar productions (Ex: E->E+T|T). Press Ctrl+D (Linux/Mac) or Ctrl+Z (Windows) to
stop:\n");
    yylex();
    int lr = check_left_recursion();
    int lf = check_left_factoring();
    if(!lr && !lf)
        printf(" Given grammar is likely LL(1)\n");
    else
        printf(" Grammar is NOT LL(1)\n");
    return 0;
}
// Fix: yywrap function added
int yywrap() {
    return 1;
}
```

```
C:\Users\shahu>flex "C:\Users\shahu\Music\pr8.l.txt"

C:\Users\shahu>gcc lex.yy.c -o pr8.exe

C:\Users\shahu>pr8.exe
Enter grammar productions (Ex: E->E+T|T). Press Ctrl+D (Linux/Mac) or Ctrl+Z (Windows) to stop:
S->aABb
A->c|#
B->d|#
^Z
Γ£à Given grammar is likely LL(1)

C:\Users\shahu>flex "C:\Users\shahu\Music\pr8.l.txt"

C:\Users\shahu>gcc lex.yy.c -o pr8.exe

C:\Users\shahu>pr8.exe
Enter grammar productions (Ex: E->E+T|T). Press Ctrl+D (Linux/Mac) or Ctrl+Z (Windows) to stop:
S->A
A->aB|aC|aD|Ac
B->bBC|f
C->g|#

D->d|#
^Z
Γ¥î Left recursion detected in: A->aB|aC|aD|Ac
Γ¥i Left factoring needed in: A->aB|aC|aD|Ac
Γ¥î Grammar is NOT LL(1)
```

Date :

Practical NO. 9

* Aim :- Write a lex program to check whether given LR(0) or not

* Theory :-

LR(1) parsing technique belongs to class of bottom up parser which mean it attempt to build parser three starting from tokens and working of to root.

→ LR(1) is checked by constructing canonical LR(1) collection of items and LR(1) parsing table then verifying that action table contains no conflict.

* Result :- Hence program are executed successfully.

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

#define MAX 50

typedef struct {
  char left;
  char right[MAX];
} production;

production prod[MAX];
int n = 0;

int has_conflict(char lhs);
int has_left_recursion();
%}

%%
[A-Z]->[A-Za-z|]+ {
  char *arrow = strstr(yytext, "->");
  prod[n].left = yytext[0];
  strcpy(prod[n].right, arrow + 2);
  n++;
}
[ \t\n]+ ;
. ;
%%

int main() {
```

```
        return 0;
    }

    int has_conflict(char lhs) {
        // Naive check: if same LHS has multiple terminals starting same
        char first_chars[MAX];
        int count = 0;

        for (int i = 0; i < n; i++) {
            if (prod[i].left == lhs) {
                char start = prod[i].right[0];
                for (int j = 0; j < count; j++) {
                    if (first_chars[j] == start)
                        return 1;
                }
                first_chars[count++] = start;
            }
        }
    return 0;
}

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

```
E:\Compiler Design>lr1_check
Enter CFG productions (e.g. E->E+T|T). Press Ctrl+Z or Ctrl+D to end:
E->E+T|T
T->T*F|F
F->(E)|id

^D
^Z

Total productions: 2
Grammar is NOT LR(1): Left recursion detected.
```

Date :

Practical No.10

* Aim :- Write a program to generate Three address code.

* Theory :-

Three address code (TAC) is an intermediate code used by compiler where each instruction has at most three operands- thus for operation and one for operand result.

* formula :- result = operand1 of operand2

- Types :

- quadruples :- (OP, arg1, arg2, result)
- Triples :- (op, arg2, arg2) result is implified by position.

* Indirect triple : Uses pointer to triples.

* Result :- Hence a program are executed successfully.

Practical 10

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

int tempCount = 1;
char temp[5];

// Function to create new temporary variable
void newTemp(char *t) {
   sprintf(t, "t%d", tempCount++);
}

%}

%%
[a-zA-Z]    { printf("PUSH %s\n", yytext); }
[0-9]+      { printf("PUSH %s\n", yytext); }
[\+\-\*/]   {
              char op = yytext[0];
              char t[5];
              newTemp(t);
              printf("POP R2\n");
              printf("POP R1\n");
              printf("%c R1, R2\n", op);
              printf("PUSH %s\n", t);
              printf("%s = R1 %c R2\n", t, op);
            }
\n          { printf("\n"); }
.           ;
%%

int main() {
   printf("Enter arithmetic expression: ");
   yylex();
   return 0;
}
```

Output:

```
C:\Users\shahu>win_flex three_address.l

C:\Users\shahu>gcc lex.yy.c -o three_address.exe

C:\Users\shahu>three_address.exe
Enter arithmetic expression: a+b*c
PUSH a
PUSH b
PUSH c
POP R2
POP R1
* R1, R2
PUSH t1
t1 = R1 * R2
POP R2
POP R1
+ R1, R2
PUSH t2
t2 = R1 + R2
```