LEX Program

**1) To count vowels and consonants**

%{

#include <stdio.h>

int vowel\_count = 0;

int cons\_count = 0;

%}

%%

[aAeEiIoOuU] { vowel\_count++; }

[bBcCdDfFgGhHjJkKlLmMnNpPqqrRsStTvVwWxXyYzZ] { cons\_count++; }

. { /\* Ignore other characters \*/ }

%%

int main() {

printf("Enter the text: ");

yylex();

printf("Vowels: %d\n", vowel\_count);

printf("Consonants: %d\n", cons\_count);

return 0;

}

int yywrap() {

return 1;

}  
**O/P : lex vow.l**

**gcc lex.yy.c -lfl**

**./a.out**

**CTRL+D (to generate the o/p if not shown on the screen)**  
 **2) To reverse a string**

%{

#include <stdio.h>

char a[1000];

int i = 0;

%}

%%

. { a[i++] = yytext[0]; }

\n {

a[i] = '\0';

for (int k = i - 1; k >= 0; k--) {

printf("%c", a[k]);

}

printf("\n");

i = 0;

}

%%

int main() {

yylex();

return 0;

}

int yywrap() {

return 1**;**

**}**

**3) Print Predecessor and Succesor**

%{

#include <stdio.h>

%}

%%

[a-zA-Z] {

char ch = yytext[0];

char pred = ch - 1;

char succ = ch + 1;

// Wrap around for edge cases

if (ch == 'a') pred = 'z';

if (ch == 'A') pred = 'Z';

if (ch == 'z') succ = 'a';

if (ch == 'Z') succ = 'A';

printf("Character: %c, Predecessor: %c, Successor: %c\n", ch, pred, succ);

}

.|\n ; // Ignore other characters

%%

int main() {

printf("Enter the alphabets:\n");

yylex();

return 0;

}

int yywrap() {

return 1;

}

**4) To find whether a number is divisible by 7 or not**

%{

#include <stdio.h>

int num;

%}

%%

[0-9]+ {

sscanf(yytext, "%d", &num);

if (num % 7 == 0)

printf("Number %d is divisible by 7\n", num);

else

printf("Number %d is NOT divisible by 7\n", num);

}

.|\n ; // Ignore any other character

%%

int main() {

printf("Enter a number: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}  
  
**5) To count number of letters, words, lines and blank ingiven input**

%{

#include <stdio.h>

#include <ctype.h>

int letters = 0, words = 0, lines = 0, blanks = 0;

%}

%%

[a-zA-Z]+ {

letters += yyleng;

words++;

}

[0-9]+ { words++; }

[ \t]+ { blanks++; }

\n { lines++; }

%%

int main() {

printf("Enter the input (Press Ctrl+D to end):\n");

yylex();

printf("\nLetters: %d\n", letters);

printf("Words: %d\n", words);

printf("Lines: %d\n", lines);

printf("Blanks: %d\n", blanks);

return 0;

}

int yywrap() {

return 1;

}

**6) Write a lex program to find the factorial of a given number**

%{

#include <stdio.h>

long long factorial(int n) {

if (n == 0 || n == 1)

return 1;

else

return n \* factorial(n - 1);

}

%}

%%

[0-9]+ {

int num;

sscanf(yytext, "%d", &num);

if (num < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

printf("Factorial of %d is %lld\n", num, factorial(num));

}

}

.|\n ; // Ignore everything else

%%

int main() {

printf("Enter a number: ");

yylex();

return 0;

}

int yywrap() {

return 1;

}

**7) Compound or simple statement**

%{

#include <stdio.h>

int isCompound = 0;

%}

%%

(and|but|or|yet|so|for|nor) { isCompound = 1; }

.|\n ;

%%

int main() {

printf("Enter a statement: ");

yylex();

if (isCompound)

printf("The statement is a COMPOUND statement.\n");

else

printf("The statement is a SIMPLE statement.\n");

return 0;

}

int yywrap() {

return 1;

}

**8) Regular exp a^2b^n**

%{

#include <stdio.h>

int valid = 0;

%}

%%

aab+ { valid = 1; }

.|\n ; // Ignore everything else

%%

int main() {

printf("Enter the string: ");

yylex();

if (valid)

printf("The input matches the pattern a^2b^n\n");

else

printf("The input does NOT match the pattern a^2b^n\n");

return 0;

}

int yywrap() {

return 1;

}

**9) balanced paranthepars**

%{

#include <stdio.h>

int balance = 0;

int unbalanced = 0;

%}

%%

"(" { balance++; }

")" {

if (balance > 0)

balance--;

else

unbalanced = 1;

}

.|\n ; // Ignore other characters

%%

int main() {

printf("Enter the input (Ctrl+D to end):\n");

yylex();

if (balance == 0 && unbalanced == 0)

printf("Parentheses are BALANCED.\n");

else

printf("Parentheses are NOT BALANCED.\n");

return 0;

}

int yywrap() {

return 1;

}

**Epsilon Closure**

from collections import defaultdict

def epsilon\_closure(state, epsilon\_trans, visited=None):

if visited is None:

visited = set()

visited.add(state)

for next\_state in epsilon\_trans[state]:

if next\_state not in visited:

epsilon\_closure(next\_state, epsilon\_trans, visited)

return visited

# Input

n = int(input("Enter number of states: "))

e = int(input("Enter number of epsilon transitions: "))

epsilon\_trans = defaultdict(list)

for \_ in range(e):

u, v = map(int, input("Enter epsilon transition (from to): ").split())

epsilon\_trans[u].append(v)

s = int(input("Enter state to compute epsilon-closure: "))

closure = epsilon\_closure(s, epsilon\_trans)

print(f"Epsilon-closure({s}) =", sorted(closure))

**First and Follow Function**

from collections import defaultdict

def compute\_first(grammar):

first = defaultdict(set)

def first\_of(symbol):

if not symbol.isupper():

return {symbol}

if symbol in first and first[symbol]:

return first[symbol]

result = set()

for production in grammar[symbol]:

if production == '':

result.add('ε')

else:

for sym in production:

temp = first\_of(sym)

result.update(temp - {'ε'})

if 'ε' not in temp:

break

else:

result.add('ε')

first[symbol] = result

return result

for non\_terminal in grammar:

first\_of(non\_terminal)

return first

def compute\_follow(grammar, first\_sets, start\_symbol):

follow = defaultdict(set)

follow[start\_symbol].add('$')

while True:

updated = False

for lhs in grammar:

for production in grammar[lhs]:

for i in range(len(production)):

B = production[i]

if B.isupper():

rest = production[i+1:]

temp = set()

if rest:

for sym in rest:

temp |= first\_sets[sym] - {'ε'}

if 'ε' in first\_sets[sym]:

continue

else:

break

else:

temp |= follow[lhs]

else:

temp = follow[lhs]

if not temp.issubset(follow[B]):

follow[B] |= temp

updated = True

if not updated:

break

return follow

# ----------- Main Program with User Input ------------

grammar = defaultdict(list)

n = int(input("Enter number of productions: "))

print("Enter productions (e.g., A->aB or B->ε):")

for \_ in range(n):

prod = input().strip()

lhs, rhs = prod.split("->")

lhs = lhs.strip()

alternatives = rhs.strip().split('|')

for alt in alternatives:

grammar[lhs].append(alt.strip().replace('ε', ''))

start\_symbol = list(grammar.keys())[0]

first\_sets = compute\_first(grammar)

follow\_sets = compute\_follow(grammar, first\_sets, start\_symbol)

print("\nFirst sets:")

for nt in first\_sets:

print(f"First({nt}) = {{ {', '.join(first\_sets[nt])} }}")

print("\nFollow sets:")

for nt in follow\_sets:

print(f"Follow({nt}) = {{ {', '.join(follow\_sets[nt])} }}")

O/P: Enter number of productions: 3

Enter productions (e.g., A->aB or B->ε):

A->B|a|b

B->cB|b|C

C->c  
  
First sets:

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

First(B) = { b, c }

First(C) = { c }

Follow sets:

Follow(A) = { $ }

Follow(B) = { $ }

Follow(C) = { b }