1. **Kailash Nath Reddy (191911147) Slot-C Compiler Design**

**Exp. No. 1**

**1.The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program.**

**Program:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

int main()

{

int i,ic=0,m,cc=0,oc=0,j;

char b[30],operators[30],identifiers[30],constants[30];

printf("enter the string : ");

scanf("%[^\n]s",&b);

for(i=0;i<strlen(b);i++)

{

if(isspace(b[i]))

{

continue;

}

else if(isalpha(b[i]))

{

identifiers[ic] =b[i];

ic++;

}

else if(isdigit(b[i]))

{

m=(b[i]-'0');

i=i+1;

while(isdigit(b[i]))

{

m=m\*10 + (b[i]-'0');

i++;

}

i=i-1;

constants[cc]=m;

cc++;

}

else

{

if(b[i]=='\*')

{

operators[oc]='\*';

oc++;

}

else if(b[i]=='-')

{

operators[oc]='-';

oc++;

}

else if(b[i]=='+')

{

operators[oc]='+';

oc++;

}

else if(b[i]=='=')

{

operators[oc]='=';

oc++;

}

}

}

printf(" identifiers : ");

for(j=0;j<ic;j++)

{

printf("%c ",identifiers[j]);

}

printf("\n constants : ");

for(j=0;j<cc;j++)

{

printf("%d ",constants[j]);

}

printf("\n operators : ");

for(j=0;j<oc;j++)

{

printf("%c ",operators[j]);

}

}

**Output:**

enter the string : a=2-d\*3+f

identifiers : a d f

constants : 2 3

operators : = - \* +

**Exp. No. 2**

**2.Extend the lexical Analyzer to Check comments, dened as follows in C:**

**a) A comment begins with // and includes all characters until the end of that line.**

**b) A comment begins with /\* and includes all characters through the next occurrence of the character sequence \*/Develop a lexical Analyzer to identify whether a given line is a comment or not.**

**Program:**

#include<stdio.h>

#include<conio.h>

int main()

{

char com[30];

int i=2,a=0;

printf("\n Enter comment:");

gets(com);

if(com[0]=='/')

{

if(com[1]=='/')

printf("\n It is a comment");

else if(com[1]=='\*')

{

for(i=2;i<=30;i++)

{

if(com[i]=='\*'&&com[i+1]=='/')

{

printf("\n It is a comment");

a=1;

break;

}

else

continue;

}

if(a==0)

printf("\n It is not a comment");

}

else

printf("\n It is not a comment");

}

else

printf("\n It is not a comment");

}

**Output:**

**Input:** Enter comment: //kailashdharani

**Output**: It is a comment

**Input:** Enter comment: kailashdharani

**Output**: It is not a comment

**Exp. No. 3**

**Design a lexical Analyzer for given language should ignore the redundant spaces, tabs and new lines and ignore comments using C**

**Program:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

int isKeyword(char buffer[]){

char keywords[32][10] = {"main","auto","break","case","char","const","continue","default",

"do","double","else","enum","extern","float","for","goto",

"if","int","long","register","return","short","signed",

"sizeof","static","struct","switch","typedef",

"unsigned","void","printf","while"};

int i, flag = 0;

for(i = 0; i < 32; ++i)

{

if(strcmp(keywords[i], buffer) == 0)

{

flag = 1;

break;

}

}

return flag;

}

int main()

{

char ch, buffer[15], operators[] = "+-\*/%=";

FILE \*fp;

int i,j=0;

fp = fopen("flex\_input.txt","r");

if(fp == NULL){

printf("error while opening the file\n");

exit(0);

}

while((ch = fgetc(fp)) != EOF){

for(i = 0; i < 6; ++i){

if(ch == operators[i])

printf("%c is operator\n", ch);

}

if(isalnum(ch)){

buffer[j++] = ch;

}

else if((ch == ' ' || ch == '\n') && (j != 0)){

buffer[j] = '\0';

j = 0;

if(isKeyword(buffer) == 1)

printf("%s is keyword\n", buffer);

else

printf("%s is identifier\n", buffer);

}

}

fclose(fp);

return 0;

}

**Input:** flex\_input.txt

main ( )

{

int a, b, c ,d, e;

c = b + c - e;

printf ( "%d" ,c ) ;

}

**Output:**

main is keyword

int is keyword

a is identifier

b is identifier

c is identifier

d is identifier

e is identifier

c is identifier

= is operator

b is identifier

+ is operator

c is identifier

- is operator

e is identifier

print f is keyword

% is operator

d is identifier

c is identifier

**Exp. No. 4**

**Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using regular arithmetic operators using C**

**Program:**

#include<stdio.h>

#include<conio.h>

int main()

{

char s[5];

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

gets(s);

switch(s[0])

{

case'>':

if(s[1]=='=')

printf("\n Greater than or equal");

else

printf("\n Greater than");

break;

case'<':

if(s[1]=='=')

printf("\n Less than or equal");

else

printf("\nLess than");

break;

case'=':

if(s[1]=='=')

printf("\nEqual to");

else

printf("\nAssignment");

break;

case'!':

if(s[1]=='=')

printf("\nNot Equal");

else

printf("\n Bit Not");

break;

case'&':

if(s[1]=='&')

printf("\nLogical AND");

else

printf("\n Bitwise AND");

break;

case'|':

if(s[1]=='|')

printf("\nLogical OR");

else

printf("\nBitwise OR");

break;

case'+':

printf("\n Addition");

break;

case'-':

printf("\nSubstraction");

break;

case'\*':

printf("\nMultiplication");

break;

case'/':

printf("\nDivision");

break;

case'%':

printf("Modulus");

break;

default:

printf("\n Not a operator");

}

}

**Output:**

Enter any operator: =>

Assignment

**Exp. No. 5**

**Design a lexical Analyzer to find the number of whitespaces and newline characters using C.**

**Program:**

#include <stdio.h>

int main()

{

char str[100];//input string with size 100

int words=0,newline=0,characters=0; // counter variables

scanf("%[^~]",&str);//scanf formatting

for(int i=0;str[i]!='\0';i++)

{

if(str[i] == ' ')

{

words++;

}

else if(str[i] == '\n')

{

newline++;

words++;//since with every next line new words start. corner case 1

}

else if(str[i] != ' ' && str[i] != '\n'){

characters++;

}

}

if(characters > 0)//Corner case 2,3.

{

words++;

newline++;

}

printf("Total number of words : %d\n",words);

printf("Total number of lines : %d\n",newline);

printf("Total number of characters : %d\n",characters);

return 0;

}

**Output:**

void main()

{

int a;

int b;

int c;

int d;

int e;

a = b + c;

c = d \* e;

}

Total number of words : 23

Total number of lines : 10

**Exp. No. 6**

**Develop a lexical Analyzer to test whether a given identifier is valid or not using C.**

**Program:**

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

int main()

{

char a[10];

int flag, i=1;

printf("\n Enter an identifier:");

gets(a);

if(isalpha(a[0]))

flag=1;

else

printf("\n Not a valid identifier");

while(a[i]!='\0')

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0;

break;

} i++;

}

if(flag==1)

printf("\n Valid identifier");

}

**Output:**

Enter an identifier: kailashdharani519

Valid identifier

**Exp. No. 7**

**Write a C program to find FIRST( ) - predictive parser for the given grammar**

S → AaAb / BbBa

A → ∈

B → ∈

**Program:**

#include<stdio.h>

#include<ctype.h>

void FIRST(char[],char );

void addToResultSet(char[],char);

int numOfProductions;

char productionSet[10][10];

int main()

{

int i;

char choice;

char c;

char result[20];

printf("How many number of productions ? :");

scanf(" %d",&numOfProductions);

for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T

{

printf("Enter productions Number %d : ",i+1);

scanf(" %s",productionSet[i]);

}

do

{

printf("\n Find the FIRST of :");

scanf(" %c",&c);

FIRST(result,c); //Compute FIRST; Get Answer in 'result' array

printf("\n FIRST(%c)= { ",c);

for(i=0;result[i]!='\0';i++)

printf(" %c ",result[i]); //Display result

printf("}\n");

printf("press 'y' to continue : ");

scanf(" %c",&choice);

}

while(choice=='y'||choice =='Y');

}

/\*

\*Function FIRST:

\*Compute the elements in FIRST(c) and write them

\*in Result Array.

\*/

void FIRST(char\* Result,char c)

{

int i,j,k;

char subResult[20];

int foundEpsilon;

subResult[0]='\0';

Result[0]='\0';

//If X is terminal, FIRST(X) = {X}.

if(!(isupper(c)))

{

addToResultSet(Result,c);

return ;

}

//If X is non terminal

//Read each production

for(i=0;i<numOfProductions;i++)

{

//Find production with X as LHS

if(productionSet[i][0]==c)

{

//If X → ε is a production, then add ε to FIRST(X).

if(productionSet[i][2]=='$') addToResultSet(Result,'$');

//If X is a non-terminal, and X → Y1 Y2 … Yk

//is a production, then add a to FIRST(X)

//if for some i, a is in FIRST(Yi),

//and ε is in all of FIRST(Y1), …, FIRST(Yi-1).

else

{

j=2;

while(productionSet[i][j]!='\0')

{

foundEpsilon=0;

FIRST(subResult,productionSet[i][j]);

for(k=0;subResult[k]!='\0';k++)

addToResultSet(Result,subResult[k]);

for(k=0;subResult[k]!='\0';k++)

if(subResult[k]=='$')

{

foundEpsilon=1;

break;

}

//No ε found, no need to check next element

if(!foundEpsilon)

break;

j++;

}

}

}

}

return ;

}

/\* addToResultSet adds the computed

\*element to result set.

\*This code avoids multiple inclusion of elements

\*/

void addToResultSet(char Result[],char val)

{

int k;

for(k=0 ;Result[k]!='\0';k++)

if(Result[k]==val)

return;

Result[k]=val;

Result[k+1]='\0';

}

**Output:**

How many number of productions ? :5

Enter productions Number 1 : S=KDkd

Enter productions Number 2 : S=KdkD

Enter productions Number 3 : S=DK

Enter productions Number 4 : K=%

Enter productions Number 5 : D=%

Find the FIRST of :S

FIRST(S)= { % }

press 'y' to continue : y

Find the FIRST of :K

FIRST(K)= { % }

press 'y' to continue : y

Find the FIRST of :D

FIRST(D)= { % }

press 'y' to continue : n

**Exp. No. 8**

**Write a C program to find FOLLOW( )** **- predictive parser for the given grammar**

S → AaAb / BbBa

A → ∈

B → ∈

**Program:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

int limit, x = 0;

char production[10][10], array[10];

void find\_first(char ch);

void find\_follow(char ch);

void Array\_Manipulation(char ch);

int main()

{

int count;

char option, ch;

printf("\nEnter Total Number of Productions:\t");

scanf("%d", &limit);

for(count = 0; count < limit; count++)

{

printf("\nValue of Production Number [%d]:\t", count + 1);

scanf("%s", production[count]);

}

do

{

x = 0;

printf("\nEnter production Value to Find Follow:\t");

scanf(" %c", &ch);

find\_follow(ch);

printf("\nFollow Value of %c:\t{ ", ch);

for(count = 0; count < x; count++)

{

printf("%c ", array[count]);

}

printf("}\n");

printf("To Continue, Press Y:\t");

scanf(" %c", &option);

}while(option == 'y' || option == 'Y');

return 0;

}

void find\_follow(char ch)

{

int i, j;

int length = strlen(production[i]);

if(production[0][0] == ch)

{

Array\_Manipulation('$');

}

for(i = 0; i < limit; i++)

{

for(j = 2; j < length; j++)

{

if(production[i][j] == ch)

{

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

{

find\_first(production[i][j + 1]);

}

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

{

find\_follow(production[i][0]);

}

}

}

}

}

void find\_first(char ch)

{

int i, k;

if(!(isupper(ch)))

{

Array\_Manipulation(ch);

}

for(k = 0; k < limit; k++)

{

if(production[k][0] == ch)

{

if(production[k][2] == '$')

{

find\_follow(production[i][0]);

}

else if(islower(production[k][2]))

{

Array\_Manipulation(production[k][2]);

}

else

{

find\_first(production[k][2]);

}

}

}

}

void Array\_Manipulation(char ch)

{

int count;

for(count = 0; count <= x; count++)

{

if(array[count] == ch)

{

return;

}

}

array[x++] = ch;

}

**Output:**

Enter Total Number of Productions: 4

Value of Production Number [1]: S=KaKb

Value of Production Number [2]: S=DbDa

Value of Production Number [3]: A=\*

Value of Production Number [4]: B=\*

Enter production Value to Find Follow: S

Follow Value of S: { \* }

To Continue, Press Y: y

Enter production Value to Find Follow: A

Follow Value of A: { a b }

To Continue, Press Y: y

Enter production Value to Find Follow: B

Follow Value of B: { b a }

To Continue, Press Y: n

**Exp. No. 9**

**Implement a C program to eliminate left recursion from a given CFG.**

S → (L) / a

L → L , S / S

**Program:**

#include<stdio.h>

#include<string.h>

#define SIZE 10

int main () {

char non\_terminal;

char beta,alpha;

int num;

char production[10][SIZE];

int index=3; /\* starting of the string following "->" \*/

printf("Enter Number of Production : ");

scanf("%d",&num);

printf("Enter the grammar as E->E-A :\n");

for(int i=0;i<num;i++){

scanf("%s",production[i]);

}

for(int i=0;i<num;i++){

printf("\nGRAMMAR : : : %s",production[i]);

non\_terminal=production[i][0];

if(non\_terminal==production[i][index]) {

alpha=production[i][index+1];

printf(" is left recursive.\n");

while(production[i][index]!=0 && production[i][index]!='|')

index++;

if(production[i][index]!=0) {

beta=production[i][index+1];

printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal);

printf("\n%c\'->%c%c\'|E\n",non\_terminal,alpha,non\_terminal);

}

else

printf(" can't be reduced\n");

}

else

printf(" is not left recursive.\n");

index=3;

}

}

**Output:**

Enter Number of Production : 2

Enter the grammar as E->E-A :

S->(K)|a

K->K,S|S

GRAMMAR : : : S->(K)|a is not left recursive.

GRAMMAR : : : K->K,S|S is left recursive.

Grammar without left recursion:

K->SK'

K'->,K'|E

**Exp. No. 10**

**Implement a C program to eliminate left factoring from a given CFG.**

S → iEtS / iEtSeS / a

E → b

**Program:**

#include<stdio.h>

#include<string.h>

int main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

int i,j=0,k=0,l=0,pos;

printf("Enter Production : S->");

gets(gram);

for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i];

part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)

part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

if(part1[i]==part2[i])

{

modifiedGram[k]=part1[i];

k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){

newGram[j]=part1[i];

}

newGram[j++]='|';

for(i=pos;part2[i]!='\0';i++,j++){

newGram[j]=part2[i];

}

modifiedGram[k]='X';

modifiedGram[++k]='\0';

newGram[j]='\0';

printf("\n S->%s",modifiedGram);

printf("\n X->%s\n",newGram);

}

**Output:**

Enter Production : S->iEtS|iEtSeS|a

S->iEtSX

X->|eS|a