Ex No : 1. IDENTIFIERS, CONSTANTS.

Bharath.
Sundara Raman. J
192011038

Aim :-

To develop a lexical Analyzer to identify identifiers, constants, operators using C program.

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed

Step 4 :- Stop.

Program :-

```
# include <stdio.h>
# include <ctype.h>
int main()
{
    int i, ic=0, m, cc=0, oc=0, j;
    char b[30], operators[30], identifiers[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++;
      }
```

```
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]);
}
}
```

Output :-

enter the string : a = b + c * e + 100.

identifiers : a b c e

constants : 100.

operators : = + * +

Result :-

Thus the C program is executed to find the identifiers.

Ex No: 2. LINE IS COMMENT.

Aim :-

To develop a lexical Analyzer to identify whether a given line is a comment or not using c.

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed.

Step 4 :- Stop.

Program :-

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char com[30];
    int i=2, a=0;
    printf("\n Entes comment : ");
    gets(com);
    if (com[0] == '/').
    {
        if (com[1] == '/')
            printf("\n 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;
}
}.
```

Output :-

Input :- Enter comment : // hello.

Output :- It is a comment.

Input :- Enter comment : hello.

Output :- It is not a comment

Result :-

Thus the c program is executed to check whether the line is comment or not.

Ex No : 3. SPACES, TABS

Aim :-

To design a lexical Analyzer for given language should ignore the redundant spaces, tabs and new lines.

Algorithm :-

step 1 :- Start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- Stop.

Program :-

```
# include <stdio.h>
# include <string.h>
int isKeyword (char buffer[]) {
char keywords [32][10] = { "main", "case", "char",
  "for", "return", "goto", "switch" };
int i, flag = 0;
for (i=0; i<32; ++i) {
if (strcmp (keywords [i], buffer) == 0) {
flag = 1;
break;
}
```

```
int main () {
char ch, buffer [15], operators [] = "+-*/%=";
FILE * fp ;
int i, j = 0 ;
fp = fopen ("3lex-input . txt ", "r ");
exit (0) ;
}.
if (iskeyword (buffer) == 1).
printf ("%s is keyword \n ", buffer) ;
else
printf ("%s is identifier \n ", buffer) ;
}.
```

Input :- 3lex_input . txt .

Output :-

main is. keyword.
int is keyword.
a is identifier.
b is identifier.
c is identifier.

Result :-

Thus the C program is executed to ignore spaces, tabs and new lines.

Ex No: 4. OPERATORS

Aim :-

To design a lexical Analyzer to validate operators to recognize the operators +, -, *, / using arithmetic operators.

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed

Step 4 :- Stop.

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[y]== '=')
        printf("\n Less than or equal");
    else
        printf("\n Less than");
    break;
case '=':
    if(s[y]== '=')
        printf("\n Equal to.");
    else
        printf("\n Assignment");
    break;
default:
    printf("\n Not a operator");
}
}
```

Output :-

Enter any operator : <=

Less than or equal

Result :-

Thus the C program is executed to find the operators.

Ex. No : 5. NO. OF WHITESPACES

Aim :-

To design a lexical Analyzer to find the number of whitespaces and newline characters using C.

Algorithm :-

Step 1 :- start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- Stop.

Program :-

```
# include <stdio.h>
int main()
{
  char str[100];
  int words = 0, newline = 0, characters = 0;
  scanf("%[^~]", &str);
 for (int i = 0; str[i] != '\0'; i++)
  {
    if (str[i] == ' ')
    {
      words++;
    }
   else if (str[i] == '\n')
   {
     newline++;
```

```
    words++;
}
else if (str[i] != ' ' && str[i] != '\n')
{   characters++;
}
printf("Total number of words : %d\n", words);
printf("Total number of lines : %d\n", newline);
printf("Total number of characters : %d\n", charac
return 0;
}
```

Output :-

```
void main()
{   int a;
    int b;
    a = b+c;
    c = d*e;
}
```

Total number of words : 18

Total number of lines : 7

Result :-

Thus the C program is executed to find the number of whitespaces and newline characters.

Ex No: 6. IDENTIFIER IS VALID.

Aim :-

To develop a lexical Analyzer to test whether a given identifier is valid or not.

Algorithm :-

step 1 :- Start

step 2 :- Declare the variables.

step 3 :- perform required operations and output will be displayed.

step 4 :- Stop.

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 : abc123

Valid identifier.

Result :-

Thus the C program is executed to find the identifier is valid or not.

Exp No: 7 FIRST() - predictive parser

Aim :-

To write a C program to find FIRST() - predictive parser for the given grammar.

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed.

Step 4 :- Stop

Program :-

```
# include <stdio.h>
# include <ctype.h>
void FIRST (char [], char);
void addToResultSet (char [], char);
int main ()
{
  int i;
  char choice;
  char c;
  char result [20];
}
do
{
  printf ("\n Find the FIRST of : ");
  scanf ("%c", &c);
```

```
FIRST(result, c);
for(i=0; result[i] != '\0'; i++)
  printf("%c", result[i]);
  printf("\n");
}
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 no. of productions ? : 4

Enter productions Number 1 : S = AaAb

Enter productions Number 2 : S = BbBa

Enter productions Number 3 : A = $

Enter productions Number 4 : B = $

Find the FIRST of : S

FIRST(S) = { $ a b }

press 'y' to continue

Find the FIRST of : A

FIRST(A) = { $ }

press 'y' to continue

Find the FIRST of : B

FIRST(B) = { $ }

press 'y' to continue

Result :-

Thus the C program is executed to find - FIRST() - predictive parser.

Ex No : 8 FOLLOW - predictive parser.

Aim :-

To write a c program to find FOLLOW() - predictive parser for the given grammar.

Algorithm :-

step 1 :- Start.

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- stop.

Program :-

```
# include <stdio.h>
# include <ctype.h>
void Array_Manipulation (char ch);
int main () {
  int count;
  for (count = 0; count < limit ; count ++).
  {
     printf ("\n Value of Production Number [%d]
        : \t", count+1) ;
     scanf ("%s", production [count]);
}
```

```
void Array- Manipulation (char ch)
{
    Pnt count;
    for (count = 0; count <= x; count++)
    {
        if (array [count] == ch)
        {
            retuen;
        }
    }
    array [x++] = ch;
}
```

Output :-

```
Enter Total Number of Productions : 4
Value of Production Number [1] : S = AaAb
Value of Production Number [2] : S = BbBa
Value of Production Number [3] : A = $
Value of Production Number [4] : B = $
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
Enter production
Value to find
Follow : B
Follow Value of B
```

Result :-

Thus the C-program is executed to find - FOLLOW() - predictive parser.

Ex.No:9. LEFT RECURSION.

Aim :-

To implement a C program to eliminate left recursion from a given CFG.

Algorithm :-

step 1 :- Start.

step 2 :- Declare the variables.

step 3 :- perform required operations and output will be displayed.

step 4 :- Stop.

Program :-

```
# include <stdio.h>
# include <string.h>
# define SIZE 10.

int main() {
  char non_terminal, beta, alpha;
  int num;
  int index =3;
  printf ("Enter Number of Production :");
  scanf ("%d", &num);
  for (int i=0; i<num; i++) {
    scanf ("%s", production[i]);
  }
```

```
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");
}
```

Output :-

Enter Number of Production : 2

Enter the grammar as E → E-A :

S → (L) | a

L → L,S | S

GRAMMAR ::: L → L,S | S is left recursive.

Grammar without left recursion :

L → SL'

L' → ,L' | E.

Result :-

Thus the C program is executed to eliminate left recursion from a given CFG.

Ex No : 10 LEFT FACTORING

Aim :-

To implement a C program to eliminate left factoring from a given CFG.

Algorithm :-

Step 1 :- start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed.

Step 4 :- stop.

Program :-

```
# include <stdio.h>
# include <string.h>
int main()
{
  char gram[20], part1[20], part2[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;
  }
}
```

Output :-

Enter Production : S → iEtS | iEtSeS | a

S → iEtSX

X → |eS|a

Result :-

Thus the C program is executed to eliminate left factoring from a given CFG.

Ex No : 11 SYMBOL TABLE OPERATIONS

Aim :-

To implement a c program to perform symbol table operations.

Algorithm :-

step 1 :- Start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- Stop

Program :-

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int cnt=0;
struct symtab
{
   char label[20];
   int addr;
}
int main()
{
 int ch, val;
 char lab[10];
```

```
do.
{
  printf("\n 1. insert \n 2. display \n 3. search \n
     4. modify \n 5. exit \n");
  scanf("%d", &ch);
  switch (ch).
  {
     case 1:
        insert();
        break;
  }.
void display.()
{
   int i;
   for.(i=0; i<cnt; i++).
   printf("%s\t %d \n", sy[i].label, sy[i].addr
}.
```

Output :-

1. Insert.
2. display.
3. search.
4. modify.
5. exit.

1

enter the label. a

enter the address 100.

Result :-

Thus the C program to perform symbol table operations.

Ex No : 12 RECURSIVE DESCENT PARSING

Aim :-

To write a C program to construct recursive descent parsing for the given grammar.

Algorithm :-

Step 1 :- Start.

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed.

Step 4 :- stop.

Program :-

```
# include <stdio.h>.
char input [100];
int i, l;
void main().
{.
  if (E())
  {.
    if(input [i+1] == '\0')
    printf ("\n String is accepted ");
    else.
    printf ("\n String is not accepted ");
    }.
  else.
  printf ("\n String not accepted ");
  getch();
}
```

```
if (input[i] == ')').
{
  i++;
  return (1);
}
else.
return (0);
}.
else.
return (0);
}.
```

Output :-

Recursive descent parsing for the following grammar.

E → TE'

E' → +TE'/@

T → FT'

T' → *FT'/@

F → (E)/ID.

Enter the string to be checked:
(a+b)*c
string is accepted.

Enter the string to be checked:
a/c+d
string is not accepted.

Result :-

Thus the c program to construct recursive descent parsing for the given grammar.

Ex No: 13 TOP DOWN PARSING TECHNIQUE.

Aim :-

To implement a C program for Top down parsing technique or Bottom up parsing technique.

Algorithm :-

step 1 :- start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- stop.

Program :-

```
# include <stdio.h>
int main () {
    char string [50];
    int flag, count = 0;
    printf ("The grammar is : S→aS, S→Sb,
                              S→ab\n");
    printf ("Enter the string to be checked : \n");
    gets (string);
    if (string [0] == 'a') {
        flag = 0;
        for (count = 1 ; string [count-1] != '\0'; count++)
        { if (string [count] == 'b') {
            flag = 1;
```

```
        continue;
    }
    else if((flag==1)&&(string[count]=='a'))
    {
        printf("The string does not belong to the
        specified grammar");
        break;
    }
    else{
        printf("string accepted");
    }
}
```

Output :-

The grammar is : $S \rightarrow aS$, $S \rightarrow Sb$, $S \rightarrow ab$.

Enter the string to be checked :

abb.

String accepted

Result :-

Thus the C program to implement Top down or bottom up parsing technique is executed.

Ex No: 14 SHIFT REDUCE PARSING

Aim :-

To implement the concept of shift reduce parsing in C programming

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed.

Step 4 :- Stop.

Program :-

```
#include<stdio.h>
#include<conio.h>
int main()
{
 printf("\n.GRAMMER\n");
 printf("\n enter the input symbol:\t");
 gets(ip_sym);
 temp[0] = ip_sym[ip_ptr]; temp[1] = '\0';
 strcat(act,temp); len = strlen(ip_sym);
 for(i=0; i<=len-1; i++)
 {
   stack[st_ptr] = ip_sym[ip_ptr];
   stack[st_ptr+1] = '\0'; ip_sym[ip_ptr]=' '; ip_ptr++
```

```
st_ptr++;
}
if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E
  ||(!strcmpi(stack,"E*E")))
{
 strcpy(stack,"E"); st_ptr=0; if(!strcmpi(stack,"E
 printf("\n $%s\t\t%s$\t\t\tE->E+E", stack, ip_sym
}
if(flag==0)
{
 printf("\n%s\t\t\t%s\t\treject", stack, ip_sym);
 exit(0);
}
return;
}
```

Output :-

enter the input symbol : a+b

stack implementation table

stack	input symbol	action
$	a+b$	-
$a	+b$	shift a
$E	+b$	E->a

Result :-

Thus the C program is implemented for shift reduce parser.

Ex No :15. OPERATOR PRECEDENCE PARSING.

Aim :-

To write a C program to implement the operator precedence parsing.

Algorithm :-

step 1 :- Start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed.

step 4 :- stop.

Program :-

```
# include <stdio.h>
# include <string.h>
char *input;
int i=0;
int top=0, l;
char prec [9][9] = {};
int getindex (char c)
{
  switch (c)
  {
    case '+': return 0;
    case '-': return 1;
    case '*': return 2;
    case '/': return 3;
```

```
case '^': return 4;
case 'i': return 5;
case '(': return 6;
case ')': return 7;
case '$': return 8;
int shift().
{
stack[++top] = *(input+ i++);
stack[top+1] = '0';
}.
```

Output :-

Enter the string. i*(i+i)*i

STACK.	INPUT.	ACTION.
$i	*(i+i)*i$	shift.
$E.	*(i+i)*i$	Reduced : E→i.
$E*.	(i+i)*i$	shift.
$E*(	i+i)*i$	shift.
$E*(i	+i)*i$	shift.

Result :-

Thus the c program is executed for operator precedence parsing.

Ex No: 16. THREE ADDRESS CODE.

Aim :-

To write a C program to Generate the Theree address code representation for the given statement.

Algorithm :-

step 1 :- Start.

step 2 :- Declare the variables.

step 3 :- perform required operations and output will be displayed.

step 4 :- stop.

Program :-

```
# include <stdio.h>.
# include <conio.h>.
struct three.
{
 char data [10], temp [7];
} s[30];
Int main() {.
 char d1 [7], d2 [7] = "t";
 int i=0; j=1, len=0;
 FILE *f1, *f2;
 clrscr();
 f1 = fopen ("sum.txt", "r");
 f2 = fopen ("out.txt", "w");
```

```
for(i=4; i<len-2; i+=2).
{
  itoa(j, d1, 7);
  strcat(d2, d1);
  strcpy.(s[j].temp, d2);
  strcpy(d1," ");
  strcpy(d2,"t");
  j++;
}.
fclose(f1);
fclose(f2);.
getch();
}.
```

Input : sum.txt.

out = in1 + in2 + in3 - in4.

Output : out.txt.

t1 = in1 + in2.

t2 = t1 + in3.

t3 = t2 - in4.

out = t3.

Result :-

Thus the c program is executed for three address code instruction.

Ex No: 17. LEXICAL ANALYZER.

Aim :-

To write a C program for implementing a Lexical Analyzer to Scan and Count the number of characters, words and lines in a file.

Algorithm :-

Step 1 :- Start.

Step 2 :- Declare the variables.

Step 3 :- perform required operations and output will be displayed.

Step 4 :- Stop.

Program :-

```
#include<stdio.h>
int main()
{
  char str[100];
  int words=0, newline=0, characters=0;
  scanf("%[^~]", &str);
  for(int i=0; str[i]!='\0'; i++)
  {
    if(str[i]==' ')
    {
      words++;
    }
  else if(str[i]=='\n')
  {
    newline++;
  } words++;
```

```
    else if (str[i]!=' ' && str[i]!='\n') {
        characters ++;
    }
}
if (characters > 0)
{
    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;
    a = b + c;
    c = d * e;
}
```

Total number of words : 18

Total number of lines : 7.

Result :-

Thus the C program is executed to perform lexical analyzer operation.

Ex No: 18 BACK END OF COMPILER.

Aim :-

To write a C program to implement the back end of the compiler.

Algorithm :-

Step 1 :- Start.

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed

Step 4 :- Stop.

Program :-

```
# include <stdio.h>
# include <conio.h>
int main()
{
    int n, i, j;
    char a[50][50];
    printf("enter the no: intermediate code");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
      printf("enter the 3 address code: %d:", i+1);
      for(j=0; j<6; j++)
      {
        scanf("%c", &a[i][j]);
      }
}
```

```
printf("the generated code is :");
for(i=0; i<n; i++)
{
    printf("\n mov %c, R %d", a[i][3], i);
    if(a[i][4] == '-')
    {
        printf("\n sub %c, R %d", a[i][5], i);
    }
    if(a[i][4] == '+')
    {
        printf("\n add %c, R %d", a[i][5], i);
    }
    printf("\n mov R %d, %c", i, a[i][1]);
    printf("\n");
}
return 0;
}
```

Output :-

```
enter the no. intermediate code : 2.
enter the 3 address code 1 a=b+c.
enter the 3 address code 2 d=n*d.

mov b, R0.
add c, R0.
mov R0, a.
mov n, R1.
mul d, R1
mov R1, d.
```

Result :-

Thus the C program is executed to perform back end of the compiler.

Ex No : 19 LEADING() - operator precedence

Aim :-

To write a C program to compute LEADING() - operator precedence parser for the given grammar.

Algorithm :-

Step 1 :- Start

Step 2 :- Declare the variables

Step 3 :- perform required operations and output will be displayed

Step 4 :- Stop.

Program :-

```
#include<stdio.h>
char prod[] = "EETTFF";
char res[6][3] = {{'E','+','T'};
char stack[5][2];
int top = -1;
void install (char pro, char re) {
  int i;
  for(i=0; i<18; i++) {
    if(arr[i][0] == pro && arr[i][1] == re) {
        arr[i][2] = 'T';
        break;
    }
  }
```

```
int main() {
 int i=0, j;
 char pro, re, pri='';
 for(i=0; i<6; i++) {
  for(j=0; j<3 && res[i][j] != '\0'; j++) {
    while(top>=0) {
      pro = stack.[top][0];
      re = stack[top][1];
      --top;
    }
  }
  if(arr[i][2]=='T')
    printf("%c", arr[i][1]);
 }
 getch();
}
```

Output :-

E+ T.	F * F.	T (T.
E * T.	F (T.	T) F.
E (T.	F) F	T i T.
E) F	F i T.	T $ F.
E i T	F $ F.	
E $ F.	T + F.	E → +*(i
F + F.	T * T.	F → (i
		T → *(i

Result :-

Thus the C progeram is executed is for LEADING() - operator precedence.

Ex No: 20 TRAILING - operator precedence

Aim :-

To write a C program to compute TRAILING() - operator precedence.

Algorithm :-

step 1 :- Start

step 2 :- Declare the variables

step 3 :- perform required operations and output will be displayed

step 4 :- Stop.

Program :-

```
#include<stdio.h>
char prod[6] = "EETTFF";
char res[6][3] = {{'E','+','T'},{'i','\0','\0'}};
int top = -1;
void install(char pro, char re){
 int i;
 for(i=0; i<18; i++){
  if(arr[i][0] == pro && arr[i][1]==re){
 }
 top++;
 arr[i][2]='T';
}
int main(){
 int i=0, j;
```

```
char pro, re, pri = ' ';
for(i=0; i<6; i++) {
  for (j=2; j>=0; j--) {
    install (prod[i], res[i][j]);
    break;
  }
}
for (i=0; i<18; i++) {
  if (pri != arr[i][0]) {
    pri = arr[i][0];
    printf ("\n\t%c->", pri);
  }
  if(arr[i][2] == 'T')
    printf ("%c", arr[i][1]);
}
```

Output :-

E + F.	F + F.	T + F.
E * F.	F * F	T * F.
E (F.	F (F.	T (F.
E) F.	F) F.	T) F.
E i F.	F i F.	T i F.
E $ F	F $ F.	T $ F.

Result :-

Thus the C program is executed for TRAILING() - operator precedence.