Program for Codetantra

PROGRAM 1

Objective:Program to construct a DFA which accept the language L = {$a^n b^m$ | n mod 2=0, m≥1}

Input: a a b b b

Input: a a a b b b

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

int dfa = 0;
void start(char c)
{
   if (c == 'a') {
      dfa = 1;
   }
   else if (c == 'b') {
      dfa = 3;
   }

   else {
      dfa = -1;
```

```
    }
}

void state1(char c)
{
    if (c == 'a') {
        dfa = 2;
    }
    else if (c == 'b') {
        dfa = 4;
    }
    else {
        dfa = -1;
    }
}

void state2(char c)
{
    if (c == 'b') {
        dfa = 3;
    }
    else if (c == 'a') {
        dfa = 1;
    }
    else {
```

```
        dfa = -1;
    }
}

void state3(char c)
{
    if (c == 'b') {
        dfa = 3;
    }
    else if (c == 'a') {
        dfa = 4;
    }
    else {
        dfa = -1;
    }
}

void state4(char c)
{
    dfa = -1;
}

int isAccepted(char str[])
{
    int i, len = strlen(str);
```

```
    for (i = 0; i < len; i++) {
        if (dfa == 0)
            start(str[i]);

        else if (dfa == 1)
            state1(str[i]);

        else if (dfa == 2)
            state2(str[i]);

        else if (dfa == 3)
            state3(str[i]);

        else if (dfa == 4)
            state4(str[i]);
        else
            return 0;
    }
    if (dfa == 3)
        return 1;
    else
        return 0;
}
```

```
int main()
{
   char str[] = "aaaaaabbbb";
   if (isAccepted(str))
      printf("ACCEPTED");
   else
      printf("NOT ACCEPTED");
   return 0;
}
```

Test cases:

Input: a a b b b
Output: ACCEPTED
n = 2 (even) m=3 (>=1)

Input: a a a b b b
Output: NOT ACCEPTED
n = 3 (odd), m = 3

Input: a a a a
Output: NOT ACCEPTED
n = 4, m = 0(must be >=1)

PROGRAM 2

Objective: TO IMPLEMENT LEXICAL ANALYZER FOR ANY ARITHMETIC EXPRESSION(IDENTIFIER)

INPUT:

Enter an identifier: first

Enter an identifier:1aqw

```
#include<conio.h>
 #include<stdio.h>
 #include <ctype.h>
void main()
 {
char a[10]; int flag, i=1;
 clrscr();
 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");
getch();
}
```

TEST CASE

Input: Enter an identifier: first

Output: Valid identifier

Enter an identifier:1aqw

Not a valid identifier

PROGRAM 3

Objective: TO IMPLEMENT LEXICAL ANALYZER FOR ANY ARITHMETIC EXPRESSION(FOR OPERATOR)

INPUT:

Enter any operator: *

Enter any operator: +

Enter any operator: -

```
#include<stdio.h>
#include,conio.h>
 void main()
 {
 char s[5];
 clrscr();
 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");
 }
 getch();
}
```

Test case:

Input Enter any operator: *

Output Multiplication

Input Enter any operator: +

Output Addition

Input Enter any operator: -

Output subtraction

program 4

Develope a Lexical Analzer to recognize a few patterns in C.

Input:
int
a
=
+

Code:

```
#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>


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);

}
```

```
bool isOperator(char ch)
{
     if (ch == '+' || ch == '-' || ch == '*' ||
     ch == '/' || ch == '>' || ch == '<' ||
     ch == '=')
     return (true);
     return (false);
}

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);
}
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"))
    return (true);
    return (false);
}

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);
}

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);
```

```
}

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);
}

void parse(char* str)
{
     int left = 0, right = 0;
     int len = strlen(str);

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

     if (isDelimiter(str[right]) == true && left == right)
{
```

```
        if (isOperator(str[right]) == true)
            printf("'%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)
            printf("'%s' IS A KEYWORD\n", subStr);

        else if (isInteger(subStr) == true)
            printf("'%s' IS AN INTEGER\n", subStr);

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

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

        else if (validIdentifier(subStr) == false
```

```
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n",
subStr);
            left = right;
        }
        }
        return;
}
int main()
{
        char str[100] = "int a = b + 1c; ";

        parse(str);

        return (0);
}
```

Test Case1

‘int’ is a keyword

Test Case2

‘a’ is a valid identifier

Test Case2

‘=’ is an operator

Test Case4

‘+’ is an operator

Program 5

Write a program to eliminate left recursion from the given grammar.

INPUT:

E-EA/A

A-AT/a

T=a

```
#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;
        }
    }
```

Test Case 1

E-EA/A is left recursive

Grammar without left recursion:

E-AE'

E'-AE'/E

Test Case 2

A-AT/a is left recursive

Grammar without left recursion:

A-aA'

A'-TA'/E

Test Case 3

T=a is not left recursive.

Program 6

Objective: Write a program to eliminate left factor from the given grammar

Input: Enter Production: A->

Code:

```
 #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 : A->");
       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 A->%s",modifiedGram);
       printf("\n X->%s\n",newGram);
 }
```

Test Case 1: A-> aE+bcD|aE+eIT

Test Case 2: A → aAB | aBc | aAc

Test Case 3: A → bSSaaS | bSSaSb | bSb | a

Program 7

Objective: Write a program to compute first of non-terminals.

Input:

```
 E  -> TR
R  -> +T R| #
T  -> F Y
```

```
Y  -> *F Y | #
F  -> (E) | i
```

Code:

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[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;
    count = 8;

    // The Input grammar
    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");
```

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

// Initializing the calc_first array
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;

    // Checking if First of c has
    // already been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(c == done[kay])
            xxx = 1;

    if (xxx == 1)
        continue;

    // Function call
    findfirst(c, 0, 0);
    ptr += 1;

    // Adding c to the calculated list
    done[ptr] = c;
    printf("\n First(%c) = { ", c);
    calc_first[point1][point2++] = c;

    // Printing the First Sets of the grammar
    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;

// Initializing the calc_follow array
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;

    // Checking if Follow of ck
    // has alredy been calculated
    for(kay = 0; kay <= ptr; kay++)
        if(ck == donee[kay])
            xxx = 1;

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

    // Function call
    follow(ck);
    ptr += 1;

    // Adding ck to the calculated list
    donee[ptr] = ck;
    printf(" Follow(%c) = { ", ck);
    calc_follow[point1][point2++] = ck;

    // Printing the Follow Sets of the grammar
    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++;
    }
}

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

    // Adding "$" to the follow
    // set of the start symbol
    if(production[0][0] == c) {
        f[m++] = '$';
    }
    for(i = 0; i < 10; i++)
    {
        for(j = 2;j < 10; j++)
        {
            if(production[i][j] == c)
            {
                if(production[i][j+1] != '\0')
                {
                    // Calculate the first of the next
                    // Non-Terminal in the production
                    followfirst(production[i][j+1], i, (j+2));
                }

                if(production[i][j+1]=='\0' && c!=production[i][0])
                {
                    // Calculate the follow of the Non-Terminal
                    // in the L.H.S. of the production
                    follow(production[i][0]);
                }
            }
        }
    }
}

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

    // The case where we
    // encounter a Terminal
    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))
                {
                    // Recursion to calculate First of New
                    // Non-Terminal we encounter after epsilon
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else
                    first[n++] = '#';
            }
            else if(!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                // Recursion to calculate First of
                // New Non-Terminal we encounter
                // at the beginning
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

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

    // The case where we encounter
    // a Terminal
    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;
        }

        //Including the First set of the
        // Non-Terminal in the Follow of
        // the original query
        while(calc_first[i][j] != '!')
        {
            if(calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
```

```
            else
            {
                if(production[c1][c2] == '\0')
                {
                    // Case where we reach the
                    // end of a production
                    follow(production[c1][0]);
                }
                else
                {
                    // Recursion to the next symbol
                    // in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2+1);
                }
            }
            j++;
        }
    }
}
```