Name : champion Reg no :

EX.NO.1 a) STUDY OF LEX TOOL

AIM: To study about lexical analyser generator [LEX TOOL].

1.Introduction: The unix utility lex parses a file of characters. It uses regular expression matching; typically it is used to 'tokenize' the contents of the file. In that context, it is often used together with the yacc utility.

2 Structure of a lex file

A lex file looks like

```
...definitions...
%%
...rules...
%%
...code...
```

Here is a simple example:

```
%{
int charcount=0,linecount=0;
%}
%%
. charcount++;
\n {linecount++; charcount++;}
%%
int main()
{
yylex();
printf("There were %d characters in %d lines\n",
charcount,linecount);
return 0;
}
```

2.1 Block Diagram:

LEX PGM----- | LEX COMPILER | ----- LEX .YY.C

`1

LEX.YY.C -- | GCC | -- A.OUT

I/P TEXT---- | a.out | --- O/P TEXT

Name : champion Reg no :

Definitions All code between %{ and %} is copied to the beginning of the resulting C file.

Rules A number of combinations of pattern and action: if the action is more than a single
command it needs to be in braces.

Code This can be very elaborate, but the main ingredient is the call to yylex, the lexical
analyser. If the code segment is left out, a default main is used which only calls
yylex.

3 Definitions section

There are three things that can go in the definitions section:

C code Any indented code between %{ and %} is copied to the C file. This is typically
used for defining file variables, and for prototypes of routines that are defined in the
code segment.

Definitions A definition is very much like a #define cpp directive. For example
letter [a-zA-Z]
digit [0-9]
punct [,.:;!?]
nonblank [ˆ \t]

These definitions can be used in the rules section: one could start a rule
{letter}+ {...

State definitions If a rule depends on context, it’s possible to introduce states and incorporate
those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given

4. Rules section

The rules section has a number of pattern-action pairs.

4.1 Matched text

Name : champion Reg no :

When a rule matches part of the input, the matched text is available to the programmer as
a variable char* yytext of length int yyleng.

```
%{
int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]
%%
{letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

5 Regular expressions

```
/* Regular expressions */

White       [\t\n ]+

Letter      [A-Za-z]

digit10     [0-9] /* base 10 */

digit16     [0-9A-Fa-f] /* base 16 */

identifier  {letter}(_|{letter}|{digit10})*

int10       {digit10}+
```

The example by itself is, I hope, easy to understand, but let's have a deeper look into regular expressions.

Symbol	Meaning
x	The "x" character
.	Any character except \n
[xyz]	Either x, either y, either z
[^bz]	Any character, EXCEPT b and z
[a-z]	Any character between a and z
[^a-z]	Any character EXCEPT those between a and z
R*	Zero R or more; R can be any regular expression
R+	One R or more
R?	One or zero R (that is an optionnal R)
R{2,5}	Two to 5 R
R{2,}	Two R or more

Name : champion Reg no :

R{2} | Exactly two R
"[xyz\"foo" | The string "[xyz"foo"
{NOTION} | Expansion of NOTION, that as been defined above
in the file
\X | If X is a "a", "b", "f", "n", "r", "t", or
| "v", this represent the ANSI-C interpretation of \X
\0 | ASCII 0 character
\123 | ASCII character which ASCII code is 123 IN OCTAL
\x2A | ASCII character which ASCII code is 2A in
hexadecimal
RS | R followed by S
R|S | R or S
R/S | R, only if followed by S
^R | R, only at the beginning of a line
R$ | R, only at the end of a line
<<EOF>> | End of file

Conclusion:

Name : champion Reg no :

With the help of lexical analyser generator tool ,we can separate tokens auttomatically.

EX.NO .1 b) Token Separation using LEX

AIM: To implement Token Separation using LEX TOOL.

```
letter [A-Za-z]
digit [0-9]
operator [+-*]
%%
void |
main |
if |
do |
float |
int |
printf |
char |
for |
while        {printf("%s is a keyword\n",yytext);}
%s |
%d |
%c |
%f        {printf("%s is a data type\n",yytext);}
{digit}({digit})*          {printf("%s is a number\n",yytext);}
{letter}({letter}|{digit})*        {printf("%s is an identifier\n",yytext);}
\(      {printf("%s is open paranthesis\n",yytext);}
\)      {printf("%s is close paranthesis\n",yytext);}
\;      {printf("%s is semi colon\n",yytext);}
\.      {printf("%s is a dot operator\n",yytext);}
\=      {printf("%s is assignment operator\n",yytext);}
\{      {printf("%s is open braces\n",yytext);}
\}      {printf("%s is close braces\n",yytext);}
\/      {printf("%s is a back slash\n",yytext);}
\,      {printf("%s is a comma\n",yytext);}
\"      {printf("%s is double qoute\n",yytext);}
%%
int main(int argc,char* argv[])
{
 FILE *fp;
if((fp=fopen(argv[1],"r"))==NULL)
 {
```

Name : champion Reg no :

```
printf("FILE doesn't exist");
 }
yyin=fp;
yylex();
return 0;
}
```

Inputfile :

```
Void main()
 {
int a=10;
int b=20;
float c=a/b;
print("%f",c);
}
```

Name : champion   Reg no :

Result: The program of implementation of lexial analyser using LEX hasbeen executed successfully.

Output :

Void is a keyword
Main is a keyword
(is a open parenthesis
)   is a close parenthesis
{   is a curly bracket
int   is a keyword
a=   is a equal symbol
10   is a number
;   is a double quotes
int   is a keyword
b=   is a equal symbol
20   is a number
;   is a double quotes
float is a keyword
c=   is a equal symbol
a/   is slash symbol
(is a open parenthesis
“   is a double quotes
%f   is a identifier datatype
,   is a comma
}   is a close curly bracket

Name : champion   Reg no :

EX.NO .2

Ex.3 Evaluation of Arithmetic expression using Ambiguous Grammar(Use Lex and Yacc Tool)
E-> E+E | E-E|E*E | E/E| (E) | id

```
%{
 #include<stdio.h>
 #include"y.tab.h"
void yyerror(char *);
extern int yylval;
%}
%%
[0-9]+          {yylval=atoi(yytext); return INT;}
[-*+/\n]        {return *yytext;}
[/)/(] {return *yytext;}
.         {yyerror("syntax error");}
%%
int yywrap()
{
return 1;
}
```

Ambiguous.yacc

```
%{
 #include<stdio.h>
extern int yylex(void);
void yyerror(char *);
%}
%token INT
%%
program:
program expr '\n'          {printf("%d\n",$2);}
|
;
```

Name : champion   Reg no :

```
expr:
INT             {$$=$1;}
|expr '+' expr        {$$=$1+$3;}
|expr '-' expr        {$$=$1-$3;}
|expr '*' expr        {$$=$1*$3;}
|expr '/' expr  {$$=$1/$3;}
|'(' expr ')'   {$$=$2;}
%%
void yyerror(cha*s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
}
```

Name : champion Reg no :

Result : The program of implementation of Ambiguous using YACC and LEX hasbeen executed successfully.

Output :

2+3
5

Name : champion Reg no :

EX.NO .3

Ex.4 Evaluation of Arithmetic expression using Unmbiguous Grammar(Use Lex and Yacc Tool)

E-> E+T | E-T|T
T->T*F | T/F|F
F-> (E) | id

```
%{
 #include<stdio.h>
 #include"y.tab.h"
void yyerror(char *);
extern int yylval;
%}
%%
[0-9]+        {yylval=atoi(yytext); return INT;}
[-*+/\n\(\)]        {return *yytext;}
.               {yyerror("syntax error");}
%%
int yywrap()
{
return 1;
}
```

Unambiguous.yacc

```
%{
 #include<stdio.h>
extern int yylex(void);
void yyerror(char *);
%}
%token INT
%%
program:
program expr '\n'          {printf("%d\n",$2);}
|
;
expr:
```

Name : champion Reg no :

```
T               {$$=$1;}
|expr '+' T     {$$=$1+$3;}
|expr '-' T     {$$=$1-$3;}
;
T:
F               {$$=$1;}
|T '*' F        {$$=$1*$3;}
|T '/' F        {$$=$1/$3;}
;
F:
INT             {$$=$1;}
|'(' expr ')'   {$$=$2;}
%%
void yyerror(char *s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
}
```

Name : champion Reg no :

Result : The program of implementation of Unambiguous using YACC and LEX hasbeen executed successfully.

Output :

2+5-1
6

Name : champion   Reg no :

EX.NO .4

Ex.5 Use LEX and YACC tool to implement Desktop Calculator.

E-> E+T | E-T|T
T->T*F | T/F|F
F-> (E) | id

Lex File:

```
%option noyywrap
%{
    #include<stdio.h>
    #include"y.tab.h"
    void yyerror(char *s);
    extern int yylval;
%}
digit [0-9]
%%
{digit}+        {yylval=atoi(yytext);return NUM;}
[a-z]           {yylval=toascii(*yytext)-97;return ID;}
[A-Z]           {yylval=toascii(*yytext)-65;return ID;}
[-+*/=\n]       {return *yytext;}
\(              {return *yytext;}
\)              {return *yytext;}
.               {yyerror("syntax error");}
%%
```

Calculator.yacc

```
%{
    #include<stdio.h>
    void yyerror(char*);
    extern int yylex(void);
    int val[26];
%}
%token NUM ID
%%

S:
S E '\n'            {printf("%d\n",$2);}
| S ID '=' E '\n'   {val[$2]=$4;}
|
```

Name : champion Reg no :

```
;
E:
E '+' T          {$$=$1+$3;}
|E '-' T         {$$=$1-$3;}
|T               {$$=$1;}
T:
T '*' F          {$$=$1*$3;}
|T '/' F         {$$=$1/$3;}
|F               {$$=$1;}
F:
'(' E ')'        {$$=$2;}
|NUM             {$$=$1;}
|ID              {$$=val[$1];}
%%
void yyerror(char *s)
{
printf("%s",s);
}
int main()
{
yyparse();
return 0;
}
```

Result : The program of implementation of Simple Calculator using YACC and LEX has been executed successfully.

Output :

1+3
4
a=2
b=4
a+b
6

Name : champion   Reg no :

EX.NO.5 Recursive descent parsing

```
#include<stdio.h>
#include<conio.h>
int i=0 ,f=0;
char str[30];
void E();
void Eprime();
void T();
void Tprime();
void F();
void E()
{
printf("\nE->TE'");
T();
Eprime();
}
void Eprime()
{
if(str[i]=='+')
{
	printf("\n E'->+TE'");
	i++;
	T();
	Eprime();
	}
else if((str[i]==')') || (str[i]=='$'))
	printf("\nE'->^");
}
void T()
{
printf("\nT->FT'");
F();
Tprime();
```

Name : champion   Reg no :

```
}
void Tprime()
{
	if(str[i]=='*')
		{
		printf("\nT'->*FT'");
	i++;
		F();
		Tprime();
		}
	else if((str[i]==')')||(str[i]=='+') || (str[i]==’$’))
		printf("\nT'->^");
}
void F()
{
	if(str[i]=='a')
	{
		printf("\nF->a");
		i++;
	}
	else if(str[i]=='(')
	{
		printf("\nF->(E)");
		i++;
		E();
		if(str[i]==')')
			i++;
	}
	else
		f=1;
}
void main()
{
	int len;
	clrscr();
	printf("Enter the string: ");
	scanf("%s",str);
	len=strlen(str);
	str[len]='$';
	E();
	If((str[i]=='$')&&(f==0))
	printf("\nString sucessfully parsed!");
```

```
        else
                printf("\nSyntax Error!");
getch();
}
```

Result : The program of implementation of Recursive decent parsing hasbeen executed successfully.

Output 1

Enter the string: a+a*a$
E->TE'
T->FT'
F->a
T'->^
E'->+TE'
T->FT'
F->a
T'->*FT'
F->a
T'->^
E'->^
String sucessfully parsed!

Output 2

Enter the string: a++
E->TE'
T->FT'
F->a
T'->^
E'->+TE'
T->FT'
T'->^
E'->+TE'
T->FT'
Syntax Error!

6.Shift Reduce Parser

Name : champion   Reg no :

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int z,i,j,c;
char a[16],stk[15];
void reduce();
void main()
  {  clrscr();
     puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->a");
     puts("enter input string ");
     gets(a);
     c=strlen(a);
     a[c]='$';
     stk[0]='$';
     puts("stack \t input \t action");
     for(i=1,j=0;j<c; i++,j++)
      {
          if(a[j]=='a')
           {
             stk[i]=a[j];
             stk[i+1]='\0';
             a[j]=' ';
             printf("\n%s\t%s\tshift a",stk,a);
             reduce();
           }
          else
           {
             stk[i]=a[j];
             stk[i+1]='\0';
             a[j]=' ';
             printf("\n%s\t%s\tshift->%c",stk,a,stk[i]);
             reduce();
           }
     }
           if(a[j]=='$')
           reduce();
           if((stk[1]=='E')&&(stk[2]=='\0'))
           printf("\n%s\t%s\tAccept",stk,a);
           else
           printf("\n%s\t%s\terror",stk,a);
           getch();
  }
```

Name : champion   Reg no :

```
void reduce()
  {
     for(z=1; z<=c; z++)
     if(stk[z]=='a')
        {
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n%s\t%s\tReduce by E->a",stk,a);
        }
    for(z=1; z<=c; z++)
    if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n%s\t%s\tReduce by E->E+E",stk,a);
         i=i-2;
        }
   for(z=1; z<=c; z++)
   if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n%s\t%s\tReduce by E->E*E",stk,a);
         i=i-2;
        }
    for(z=1; z<=c; z++)
    if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n%s\t%s\tReduce by E->(E)",stk,a);
         i=i-2;
        }
  }
```

Result: The program of implementation of Shift Reduce parsing has been executed successfully.

Output:

Name : champion   Reg no :

GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->a
enter input string
a*a+a
stack   input   action

```
$a      *a+a$  shift a
$E      *a+a$  Reduce by E->a
$E*      a+a$  shift->*
$E*a      +a$  shift a
$E*E      +a$  Reduce by E->a
$E        +a$  Reduce by E->E*E
$E+        a$  shift->+
$E+a        $  shift a
$E+E        $  Reduce by E->a
$E          $  Reduce by E->E+E
$E          $  Accept
```

Name : champion Reg no :

Ex. No.7. OPERATOR PRECEDENCE PARSING

```
#include<stdio.h>
#include<string.h>
int main()
{
        char stack[20],ip[20],opt[10][10][1],ter[10];
        int i,j,k,n,top=0,col,row;

        for(i=0;i<5;i++)
         {
           stack[i]='\0';
           ip[i]='\0';
           for(j=0;j<5;j++)
           {
             opt[i][j][0]='\0';
           }
         }
         printf("Enter the no.of terminals:");
         scanf("%d",&n);
         printf("\nEnter the terminals:");
         scanf(" %s",ter);
         printf("\nEnter the table values:\n");
         for(i=0;i<n;i++)
         {
       for(j=0;j<n;j++)
        {
           printf("Enter the value for %c %c:",ter[i],ter[j]);
           scanf(" %s",opt[i][j]);
        }
```

```
        }
printf("\nOPERATOR PRECEDENCE TABLE:\n");
for(i=0;i<n;i++){printf("\t%c",ter[i]);}
printf("\n");
for(i=0;i<n;i++)
{
  printf("\n%c",ter[i]);
  for(j=0;j<n;j++)
  {
      printf("\t%c",opt[i][j][0]);

   }
 }
 stack[top]='$';
 printf("\nEnter the input string:");
 scanf(" %s",ip);
 i=0;
 printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
 printf("\n%s\t\t\t%s\t\t\t",stack,ip);
 while(i<=strlen(ip))
 {
      for(k=0;k<n;k++)
      {
       if(stack[top]==ter[k])
       row=k;
       if(ip[i]==ter[k])
       col=k;
      }
      if((stack[top]=='$')&&(ip[i]=='$'))
      {
      printf("String is accepted");
      break;
      }
      else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))
      {
       stack[++top]=opt[row][col][0];
       stack[++top]=ip[i];
       printf("Shift %c",ip[i]);
       i++;
      }
       else
       {
```

Name : champion Reg no :

```
            if(opt[row][col][0]=='>')
            {
              while(stack[top]!='<')
              --top;
              top=top-1;
              printf("Reduce");
            }
           else
            {
             printf("\nString is not accepted");
             break;
            }
          }
 printf("\n");
 for(k=0;k<=top;k++)
 printf("%c",stack[k]);
 printf("\t\t\t");
 for(k=i;k<strlen(ip);k++)
 printf("%c",ip[k]);
 printf("\t\t\t");
}
return 0;
}
```

Name : champion Reg no :

Result: The program of implementation of operator precedence parsing has been executed successfully.

OUTPUT:

Enter the no.of terminals:3

Enter the terminals:a+$

Enter the table values:

Enter the value for a a:e
Enter the value for a +:>
Enter the value for a $:>
Enter the value for + a:<
Enter the value for + +:>
Enter the value for + $:>
Enter the value for $ a:<
Enter the value for $ +:<
Enter the value for $ $:a
OPERATOR PRECEDENCE TABLE:

	a	+	$
a	e	>	>
+	<	>	>
$	<	<	a

Enter the input string:a+a$

STACK	INPUT STRING	ACTION
$	a+a$	Shift a
$<a	+a$	Reduce
$	+a$	Shift +

$<+	a$	Shift a
$<+<a	$	Reduce
$<+	$	Reduce
$	$	String is accepted

8. Implementation of 3-Address Code

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10],exp[10],exp1[10],exp2[10],id1[5],op[5],id2[5];
int main()
{

while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the
choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
```

```
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;

case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';

for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
break;
}
}
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strc
mp(op,">=")==0)||(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
```

Name : champion   Reg no :

```
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address
code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address
code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address
code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
```

Name : champion Reg no :

Result: The program of implementation of 3-Address code has been executed successfully.

OR

8. Implementation of 3-Address Code

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
	int pos;
	char op;
}k[15];
void main()
{
	clrscr();
	printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
	printf("Enter the Expression :");
	scanf("%s",str);
	printf("The intermediate code\n");
	findopr();
	explore();
	getch();
}
void findopr()
{
	for(i=0;str[i]!='\0';i++)
		if(str[i]=='=')
		{
		k[j].pos=i;
```

```
		k[j++].op='=';
		}
	for(i=0;str[i]!='\0';i++)
		if(str[i]=='/')
		{
		k[j].pos=i;
		k[j++].op='/';
		}
	for(i=0;str[i]!='\0';i++)
		if(str[i]=='*')
		{
		k[j].pos=i;
		k[j++].op='*';
		}
	for(i=0;str[i]!='\0';i++)
		if(str[i]=='+')
		{
		k[j].pos=i;
		k[j++].op='+';
		}
	for(i=0;str[i]!='\0';i++)
		if(str[i]=='-')
		{
		k[j].pos=i;
		k[j++].op='-';
		}
}
void explore()
{
	i=1;
	while(k[i].op!='\0')
	{
		fleft(k[i].pos);
		fright(k[i].pos);
		str[k[i].pos]=tmpch--;
		printf("\t%c = %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
		for(j=0;j<strlen(str);j++)
			if(str[j]!='$')
				printf("%c",str[j]);
		printf("\n");
		i++;
	}
```

```
        fright(-1);
        if(no==0)
        {
                fleft(strlen(str));
                printf("\t%s = %s",right,left);
                getch();
                exit(0);
        }
        printf("\t%s =  %c",right,str[k[--i].pos]);
        getch();
}
void fleft(int x)
{
        int w=0,flag=0;
        x--;
        while(x!= -1 &&str[x]!= '+'
&&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'&&str[x]!='/')
        {
                if(str[x]!='$'&& flag==0)
                {
                left[w++]=str[x];
                left[w]='\0';
                str[x]='$';
                flag=1;
                }
                x--;
        }
}
void fright(int x)
{
        int w=0,flag=0;
        x++;
        while(x!= -1 && str[x]!=
'+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[
x]!='/')
        {
                if(str[x]!='$'&& flag==0)
                {
                right[w++]=str[x];
                right[w]='\0';
                str[x]='$';
                flag=1;
```

```
        }
        x++;
    }
}
```

Result: The program of implementation of 3-Address code has been executed successfully.

OUTPUT:

```
            INTERMEDIATE CODE GENERATION

Enter the Expression :a=b+c*d/e
The intermediate code
    Z = d/e     a=b+c*Z
    Y = c*Z     a=b+Y
    X = b+Y     a=X
    a = X
```

9. Symbol Table

Name : champion Reg no :

```
#include<stdio.h>
#include<conio.h>
struct intermediate
{
int addr;
char label[10];
char mnem[10];
char op[10];
}res;
struct symbol
{
char symbol[10];
int addr;
}sy;
void main()
{
FILE *s1,*p1;
clrscr();
s1=fopen("inter.txt","r+");
p1=fopen("symbol.txt","w");
while(!feof(s1))
{
fscanf(s1,"%d %s %s %s",&res.addr,res.label,res.mnem,res.op);
if(strcmp(res.label,"NULL")!=0)
{
strcpy(sy.symbol,res.label);
sy.addr=res.addr;
fprintf(p1,"%s\t%d\n",sy.symbol,sy.addr);
}
}
fcloseall();
printf("symbol table created");
getch();
}
```

Result: The program of implementation of symbol table has been executed successfully.

<u>OUTPUT:</u>

Name : champion Reg no :

<u>inter.txt</u>

0 NULL START 500
500 A DS 100
600 B DC 10
610 FIRST PRINT A
612 NULL READ B
613 NULL END FIRST

<u>Symbol.txt</u>

A 500
B 600
FIRST 610

Name : champion Reg no :

Ex. No.: 10 & 11 (TAKE a PRINTOUT AND ATTACH IN RECORD ONLY)

No need to take printout for observation

JFLAP MANUAL

Aim:

Name : champion Reg no :

To construct a finite automata for the given regular expression using JFLAP.

What is JFLAP?

JFLAP program makes it possible to create and simulate automata. Learning about automata with pen and paper can be difficult, time consuming and error-prone. With JFLAP we can create automata of different types and it is easy to change them as we want. JFLAP supports creation of DFA and NFA, Regular Expressions, PDA, Turing Machines, Grammars and more.

Definition:JFLAP defines a finite automaton (FA) M as the quintuple(5-tuple) $M = (Q, \Sigma, \delta, q_s, F)$ where,

Q is a finite set of states $\{q_i \mid i$ is a nonnegative integer$\}$
Σ is the finite input alphabet
δ is the transition function, $\delta : D \rightarrow 2^Q$ where D is a finite subset of $Q \times \Sigma^*$
q_s (is member of Q) is the initial state
F (is a subset of Q) is the set of final states

Sample Exercise :

Build a NFA for the given regular expression.For example, $(a|b)^*a$

1.The Editor Window

To start a new FA, start JFLAP and click the **Finite Automaton** option from the menu.

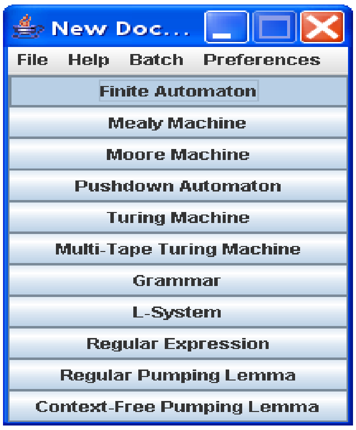


Fig.1. Starting a new FA

This brings up a new window that allows you to create and edit an FA. The editor is divided into two basic areas:

1. The canvas, which you can construct your automaton on, and
2. The toolbar, which holds the tools you need to construct your automaton.

Name : champion Reg no :

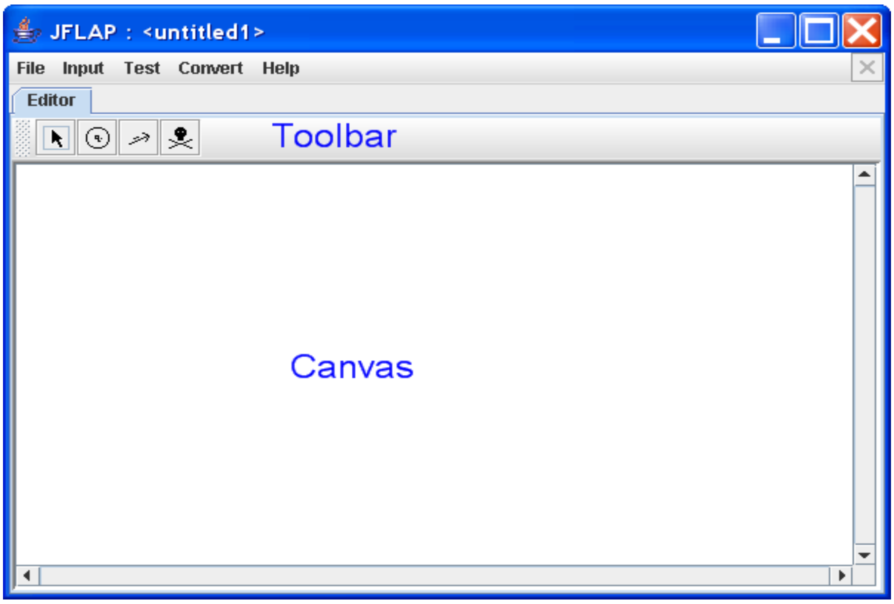


Fig.2. The editor window

Fig.3. The FA toolbar

The toolbar holds the following:

- Attribute Editor tool : sets initial and final states
- State Creator tool : creates states
- Transition Creator tool : creates transitions
- Deletor tool : deletes states and transitions

2.Creating States

- To create several states activate the State Creator tool by clicking the button on the toolbar.
- Next, click on the canvas in different locations to create states.
- The editor window should look something like this:

Name : champion Reg no :

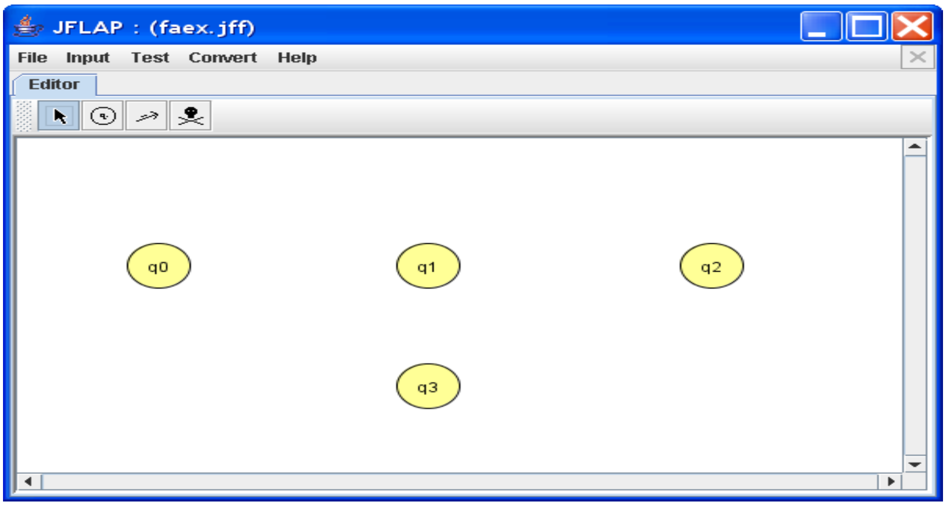


Fig.4. States created

Now that we have created our states, let's define initial and final state.

3.Defining Initial and Final States

Define q_0 to be the initial state. To define it to be our initial state, first select the Attribute Editor tool  on the toolbar, right-click on q_0. This should give a pop-up menu that looks like this:

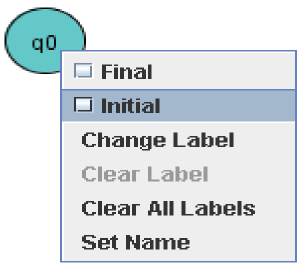
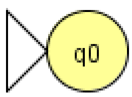


Fig.5. The state menu

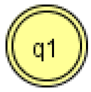
From the pop-up menu, select the checkbox **Initial**. A white arrowhead appears to the left of q_0 to indicate that it is the inital state.



q_0 defined as initial state

Next, create a final state. To define it as the final state, right-click on the state and click the checkbox **Final**. It will have a double outline, indicating that it is the final state.

Name : champion   Reg no :



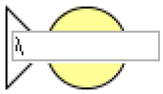
q_1 defined as final state

Now that we have defined initial and final states, let's move on to creating transitions.

4.Creating Transitions

Using the Thompsons construction generate the NFA for the given expression.

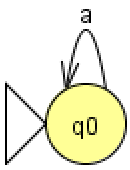
To create transition, first select the Transition Creator tool from the toolbar. Next, click on q_0 on the canvas. A text box should appear over the state:



Creating a transition

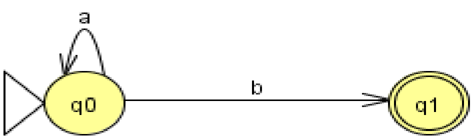
Note that λ, representing the empty string, is initially filled in. If you prefer ε representing the empty string, select **Preferences : Preferences** in the main menu to change the symbol representing the empty string.

Type "a" in the text box and press **Enter**. If the text box isn't selected, press **Tab** to select it, then enter "a". When you are done, it should look like this:



Self Transition created

To create a transition from initial state q_0 to our final state q_1, first ensure that the Transition Creator tool is selected on the toolbar. Next, click and hold on q_0, and drag the mouse to q_1 and release the mouse button. Enter "b" in the textbox.The transition between two states should look like this:



Transition between states

5.Deleting States and Transitions

Name : champion   Reg no :

To delete any state or transition , first select the Deletor tool on the toolbar. Next, click on the state or transition .

The Complete FA is now constructed for the given regular expression.

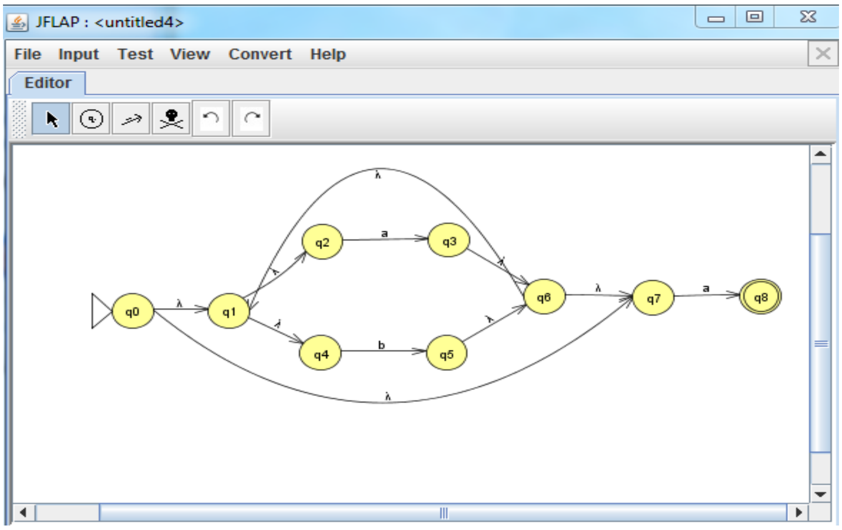


Fig.6. NFA for (a|b)*a

6.Converting to a DFA

Convert this NFA into a DFA. Click on the “Convert → Convert to DFA” menu option.

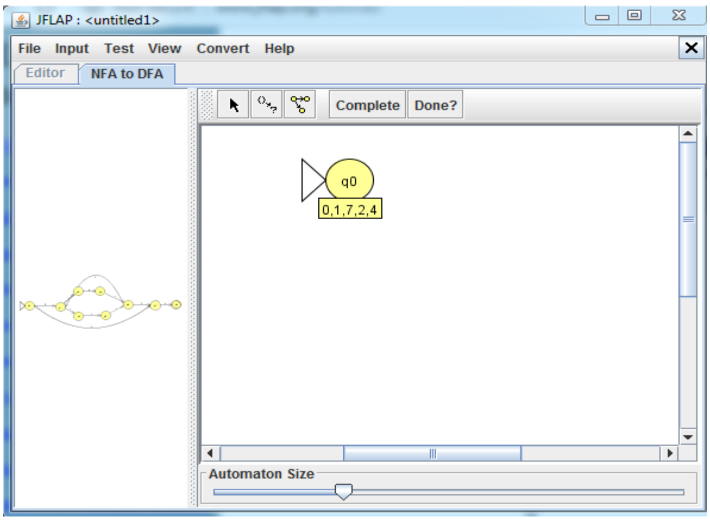


Fig. 7. Conversion step

- To see the whole DFA right away, click on the “Complete” button. Now click the “Done?” button.
- After a message informing you that the DFA is fully built, a new editor window is generated with the DFA in it. You have converted your NFA into a DFA!
- The states can be renamed by right click the states.

Name : champion   Reg no :

- If required minimization can also be done.

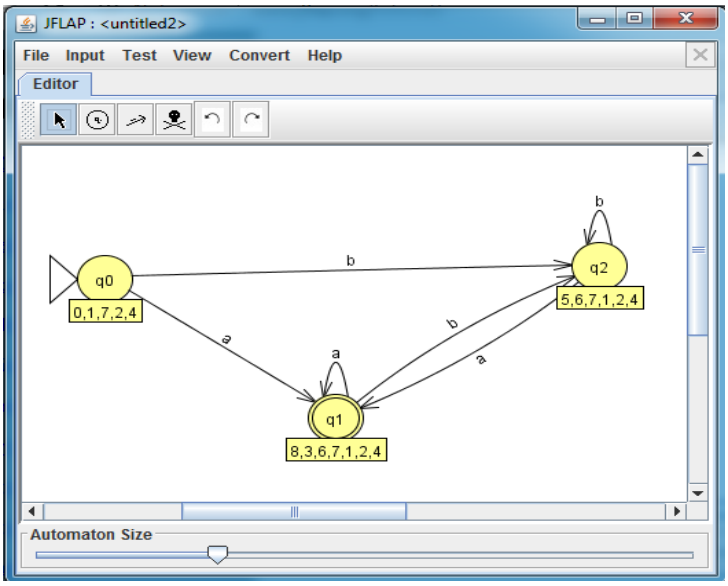


Fig. 8. DFA for(a|b)*a

7.Test Inputs

To test multiple inputs at once you can select the "Multiple Run" option. Provide the inputs and click on --> Run Inputs. The string acceptance and rejection will be displayed.

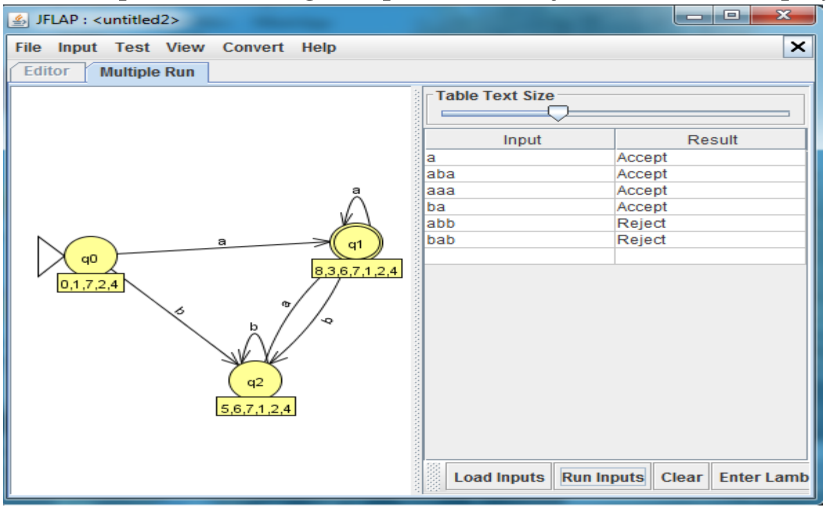


Fig.9.Test Inputs

8.Converting FA to a Grammar

When using a Finite Automaton select Convert → Convert to Grammar. The conversion view will contain your automata on the left and the grammar on the right. The "What's Left?" option will show which transition that not have been used in the grammar yet. JFLAP automatically puts labels to states to tell which symbols they represent in the grammar.

Name : champion Reg no :

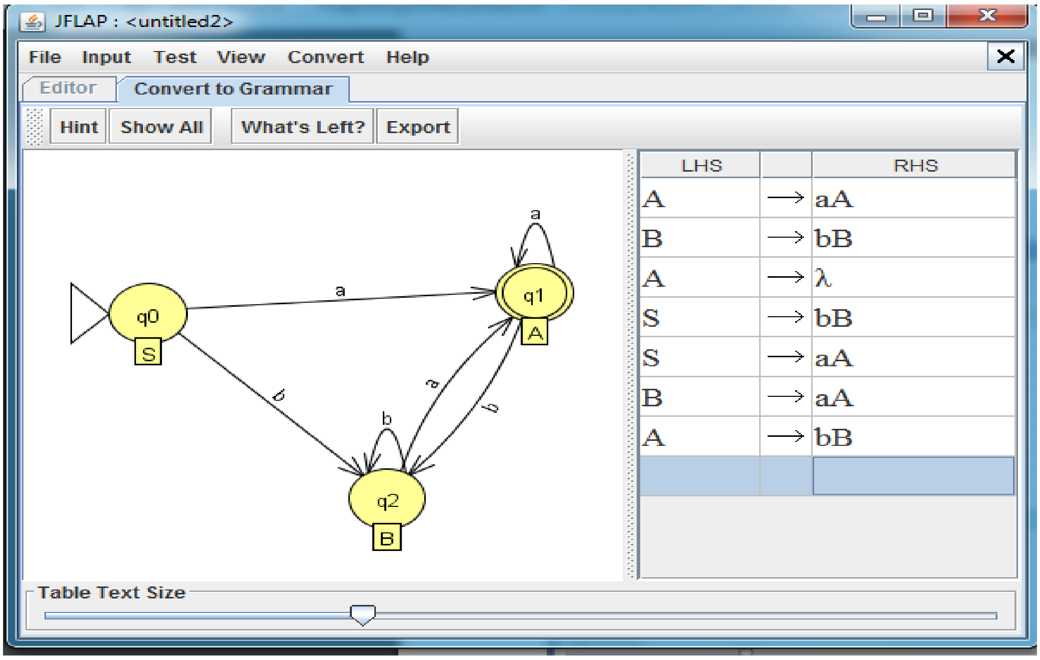


Fig. 10. FA to Grammar

Conclusion:

Thus the study on JFLAP tool used for the construction of DFA , NFA, Regular Expressions, and Grammars has been learned successfully.