

**Dhaka International University**  
**Department of Computer Science & Engineering**



**COMPILER DESIGN**  
**LAB MANUAL**

**Course Code: CSE-402**  
**Class: IV Year I Semester**

**Prepared By**  
**Md. Motiur Rahman**  
**Lecturer, Dept. of CSE**

**Dhaka International University**  
**66-Green Road, Dhaka 1205**

## COMPILER DESIGN LAB SYLLABUS

SL. No.	Experiment Name	Page No.
1	Design a lexical analyzer for given language and 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. Simulate the same in C language.	4
2	Write a C program to identify whether a given line is a comment or not.	9
3	Write a C program to recognize strings under 'a', 'a*b+', 'abb'.	11
4	Write a C program to check whether a mathematical statement is solvable or not.	14
5	Write a C program to simulate lexical analyzer for validating operators.	17
6	Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools	19
7	Write a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.	21
8	Write a C program for constructing of LL (1) parsing.	26
9	Write a C program to calculate FIRST of a regular expression.	30
10	Calculate leading for all The non-terminals of the given grammar	33
11	Design NFA, DFA, and Conversion of RE to NFA using JFAP simulations tools.	35
12	Conversion from NFA to DFA, DFA minimization using JFLAP simulation software.	53

## COMPILER DESIGN LABORATORY

### OBJECTIVE:

This laboratory course is intended to make the students experiment on the basic techniques of compiler construction and tools that can be used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object orientation.

### OUTCOMES:

Upon the completion of Compiler Design practical course, the student will be able to:

1. **Understand** the working of lex and yacc compiler for debugging of programs.
2. **Understand** and define the role of lexical analyzer, use of regular expression and transition diagrams.
3. **Understand** and use Context free grammar, and parse tree construction.
4. **Learn** & use the new tools and technologies used for designing a compiler.
5. **Develop** program for solving parser problems.
6. **Learn** how to write programs that execute faster.

### HARDWARE AND SOFTWARE REQUIREMENTS::

#### Hardware Requirements:

Processor:	Pentium I	RAM:	128MB
Hard		Disk	40 GB
Floppy Drive			1.44MB

#### Software Requirements:

Lex and Yacc tools.(A Linux Utility)

Language: C/C++

#### System Configuration on which lab is conducted

Processor:	PIV(1.8Ghz)
RAM	256MB
HDD	40GB
FDD	1.44MB
Monitor	14''Color
Keyboard	Multimedia
Operating System	Windows XP
Mouse	Scroll

## EXPERIMENT-1

### 1.1 OBJECTIVE:

Design a lexical analyzer for given language and 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. Simulate the same in C language.

### 1.2 RESOURCE:

Turbo C ++, Codeblock

### 1.3 PROGRAM LOGIC:

1. Read the input Expression
2. Check whether input is alphabet or digits then store it as identifier
3. If the input is operator store it as symbol
4. Check the input for keywords

### 1.4 PROCEDURE:

Go to debug -> run or press CTRL + F9h to run the program

### 1.5 PROGRAM:

```
#include<string.h>

#include<ctype.h>

#include<stdio.h>

void keyword(char str[10])

{

if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||str
cmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",str)=
==0||strcmp("switch",str
)==0||strcmp("case",str)==0)

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

else

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

}
```

```

main()
{

FILE *f1,*f2,*f3;

char c,str[10],st1[10];

int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;

printf("\nEnter the c program");/*gets(st1);*/

f1=fopen("input","w");

while((c=getchar())!=EOF)

putc(c,f1);

fclose(f1);

f1=fopen("input","r");

f2=fopen("identifier","w");

f3=fopen("specialchar","w");

while((c=getc(f1))!=EOF){

if(isdigit(c))

{

tokenvalue=c-'0';

c=getc(f1);

while(isdigit(c)){

tokenvalue*=10+c-'0';

c=getc(f1);

}

num[i++]=tokenvalue;

ungetc(c,f1);

}

```

```
else if(isalpha(c))

{

putc(c,f2);

c=getc(f1);

while(isdigit(c)||isalpha(c)||c=='_'||c=='$')

{

putc(c,f2);

c=getc(f1);

}

putc(' ',f2);

ungetc(c,f1);

}

else if(c==' '||c=='\t')

printf(" ");

else

if(c=='\n')

lineno++;

else

putc(c,f3);

}

fclose(f2);

fclose(f3);

fclose(f1);

printf("\nThe no's in the program are");
```

```

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

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

printf("\n");

f2=fopen("identifier","r");

k=0;
printf("The keywords and identifiers are:");
while((c=getc(f2))!=EOF){

if(c!=' ')

str[k++]=c;

else

{

str[k]='\0';

keyword(str);

k=0;
}
}

fclose(f2);

f3=fopen("specialchar","r");

printf("\nSpecial characters are");

while((c=getc(f3))!=EOF)

printf("%c",c);

printf("\n");

fclose(f3);

printf("Total no. of lines are:%d",lineno);

}

```

## 1.6 INPUT & OUTPUT:

### Input:

Enter Program \$ for termination:

```
{
int a[3],t1,t2;
t1=2; a[0]=1; a[1]=2; a[t1]=3;
t2=-(a[2]+t1*6)/(a[2]-t1);
if t2>5 then

print(t2);
else {
int t3;
t3=99;

t2=-25;
print(-t1+t2*t3);           /* this is a comment    on 2 lines */

} endif
}
$
```

### Output:

Variables : a[3] t1 t2 t3

Operator : - + \* / >

Constants : 2 1 3 6 5 99 -25

Keywords : int if then else endif

Special Symbols : , ; ( ) { }

Comments : this is a comment on 2 lines

## 1.7 LAB ASSIGNMENT

1. Write a program to recognize identifiers.
2. Write a program to recognize constants.
3. Write a program to recognize keywords and identifiers.
4. Write a program to ignore the comments in the given input source program.



## EXPERIMENT-2

### 2.1 OBJECTIVE:

Write a C program to identify whether a given line is a comment or not.

### 2.2 RESOURCE: Turbo C++, Codeblock

### 2.3 PROGRAM LOGIC:

Read the input string. Check whether the string is starting with '/' and check next character is '/' or '\*'. If condition satisfies print comment. Else not a comment.

### 2.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

### 2.5 PROGRAM:

```
#include<stdio.h>

#include<conio.h>

void main()    {

char com[30];

int i=2,a=0;

clrscr();

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");

getch(); }

```

## 2.6 INPUT & OUTPUT:

**Input:** Enter comment: //hello

**Output:** It is a comment

**Input:** Enter comment: hello

**Output:** It is not a comment

## 2.7 LAB ASSIGNMENT

1. Write a program to recognize comments and how many letters are in this comments.
2. Write a program to recognize the types of comments.
3. Write a program to find the number of line where the comments are written.

## EXPERIMENT-3

### 3.1 OBJECTIVE:

Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'.

### 3.2 RESOURCE:

Turbo C++, Codeblock

### 3.3 PROGRAM LOGIC:

By using transition diagram we verify input of the state.

If the state recognize the given pattern rule.

Then print string is accepted under a\*/ a\*b+/ abb.

Else print string not accepted.

### 3.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

### 3.5 PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
    char s[20],c;
    int state=0,i=0;
    clrscr();
    printf("\n Enter a string:");
    gets(s);
    while(s[i]!='\0')
    {
        switch(state)
        {
            case 0: c=s[i++];
            if(c=='a')
```

```
state=1;
else if(c=='b')
state=2;
else
state=6;
break;
case 1: c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=4;
else
state=6;
break;
case 2: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6;
break;
case 3: c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=2;
else
state=6;
break;
case 4: c=s[i++];
if(c=='a')
state=6;
```

```

else if(c=='b')
state=5;
else
state=6;
break;
case 5: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6;
break;
case 6: printf("\n %s is not recognised.",s);
exit(0);
}
}
        if(state==1)
printf("\n %s is accepted under rule 'a'",s);
else if((state==2)||(state==4))
printf("\n %s is accepted under rule 'a*b+',s);
else if(state==5)
printf("\n %s is accepted under rule 'abb'",s);
getch();
}

```

### 3.6 INPUT & OUTPUT:

#### Input :

Enter a String: aaaabbbbb

#### Output:

aaaabbbbb is accepted under rule 'a\*b+'

Enter a string: cdgs

cdgs is not recognized

### 3.7 LAB ASSIGNMENT

1. Write a program to recognize string under rule  $a^+b^*c^+$ .
2. Write a program to recognize string under rule  $a^*b^+c^*$ .

### EXPERIMENT -4

#### 4.1 OBJECTIVE:

Write a C program to check whether a mathematical statement is solvable or not.

#### 4.2 RESOURCE:

Turbo C++, Codeblock

#### 4.3 ALGORITHM

Start.

Declare two character arrays str[], token[] and initialize integer variables a=0, b=0, c, d.

Input the string from the user.

Repeat steps 5 to 12 till str[a] == '\0'.

If str[a] == '(' or str[a] == '{' then token[b] = '4', b++.

If str[a] == ')' or str[a] == '}' then token[b] = '5', b++.

Check if isdigit(str[a]) then repeat steps 8 till isdigit(str[a])

a++.

a--, token[b] = '6', b++.

If str[a] == '+' then token[b] = '2', b++.

If (str[a] == '\*') then token[b] = '3', b++.

a++.

token[b] = '\0';

then print the token generated for the string .

b=0.

Repeat step 22 to 31 till token[b] != '\0'

c=0.

Repeat step 24 to 30 till (token[b] == '6' and token[b+1] == '2' and token[b+2] == '6') or

(token[b] == '6' and token[b+1] == '3' and token[b+2] == '6') or (token[b] == '4' and

token[b+1] == '6' and token[b+2] == '5') or (token[c] != '\0').

token[c] = '6';

c++;

Repeat step 27 to 28 till token[c] != '\0'.

token[c] = token[c+2].

c++.

token[c-2] = '\0'.

print token.

b++.

Compare token with 6 and store the result in d.

If d=0 then print that the string is in the grammar.

Else print that the string is not in the grammar.

Stop.

#### 4.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

#### 4.5 PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
int a=0,b=0,c;
char str[20],tok[11];
clrscr();
printf("Input the expression = ");
gets(str);
while(str[a]!='\0')
{
if((str[a]=='(')||(str[a]=='{'))
{
tok[b]='4';
b++;
}
if((str[a]==')')||(str[a]=='}'))
{
tok[b]='5';
b++;
}
if(isdigit(str[a]))
{
while(isdigit(str[a]))
{
a++;
}
a--;
tok[b]='6';
b++;
}
if(str[a]=='+')
{
tok[b]='2';
b++;
}
if(str[a]=='*')
{
tok[b]='3';
b++;
}
a++;
}
```

```

tok[b]='\0';
puts(tok);
b=0;
while(tok[b]!='\0')
{
if(((tok[b]=='6')&&(tok[b+1]=='2')&&(tok[b+2]=='6'))||((tok[b]=='6')&&(tok[b+1]
)=='3')&&(tok[b+2]=='6'))||((tok[b]=='4')&&(tok[b+1]=='6')&&(tok[b+2]=='5'))/*||((tok[
b
]!='6')&&(tok[b+1]!='0'))*/)
{
tok[b]='6';
c=b+1;
while(tok[c]!='\0')
{
tok[c]=tok[c+2];
c++;
}
tok[c]='\0';
puts(tok);
b=0;
}
else
{
b++;
puts(tok);
}
}
int d;
d=strcmp(tok,"6");
if(d==0)
{
printf("It is in the grammar.");
}
else
{
printf("It is not in the grammar.");
}
getch();
}

```

#### 4.6 INPUT/OUTPUT

Input the expression = (23+)

4625

4625

4625

4625

4625

It is not in the grammar.

Input the expression = (2+(3+4)+5)

46246265265



46246265265  
46246265265  
46246265265  
46246265265  
462465265  
462465265  
462465265  
462465265  
4626265  
4626265  
46265  
46265  
465  
6  
6  
It is in the grammar.

#### **4.7 LAB ASSIGNMENT**

1. Write a program to convert  $a+(b-c)$  to prefix operation.
2. Write a program to convert  $a+(b-c)$  to postfix operation.

### **EXPERIMENT-5**

#### **5.1 OBJECTIVE:**

\*Write a C program to simulate lexical analyzer for validating operators.

#### **5.2 RESOURCE:**

Turbo C++

#### **5.3 PROGRAM LOGIC :**

Read the given input.

If the given input matches with any operator symbol.

Then display in terms of words of the particular symbol.

Else print not a operator.

#### **5.4 PROCEDURE:**

Go to debug -> run or press CTRL + F9 to run the program.

## 5.5 PROGRAM:

```
#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("\n Less than");
            break;
        case '=': if(s[1]=='=')
            printf("\n Equal to");
            else
            printf("\n Assignment");
            break;
        case '!': if(s[1]=='=')
            printf("\n Not Equal");
            else
            printf("\n Bit Not");
            break;
        case '&': if(s[1]=='&')
            printf("\n Logical AND");
            else
            printf("\n Bitwise AND");
            break;
        case '|': if(s[1]=='|')
            printf("\n Logical OR");
            else
            printf("\n Bitwise OR");
            break;
        case '+': printf("\n Addition");
            break;
        case '-': printf("\n Substraction");
            break;
    }
```

```

        case '*': printf("\nMultiplication");
                break;
        case '/': printf("\nDivision");
                break;
        case '%': printf("Modulus");
                break;
        default: printf("\n Not a operator");
        }
        getch();
}

```

## 5.6 INPUT & OUTPUT:

### Input

Enter any operator: \*

### Output

Multiplication

## EXPERIMENT-6

### 6.1 OBJECTIVE:

Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

### 6.2 RESOURCE:

Linux using Putty

### 6.3 PROGRAM LOGIC:

Read the input string.

Check whether the string is identifier/ keyword /symbol by using the rules of identifier and keywords using LEX Tool

### 6.4 PROCEDURE:

Go to terminal .Open vi editor ,Lex lex.l , cc lex.yy.c , ./a.out

### 6.5 PROGRAM:

```

/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |float |char |double |while |for |do |if |break |continue |void |switch |case |long |struct
|const |typedef |return

```

```

|else |goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
/*{printf("\n\t%s is a COMMENT\n",yytext);}*/
"*/" {COMMENT = 0;}
/* printf("\n\t%s is a COMMENT\n",yytext);}*/
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
{ {if(!COMMENT) printf("\n BLOCK
BEGINS");} } {if(!COMMENT)
printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s
IDENTIFIER",yytext);} ".*\n" {if(!COMMENT) printf("\n\t%s is
a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a
NUMBER",yytext);} {if(!COMMENT)
printf("\n\t");ECHO;printf("\n");}
( ECHO;
{if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
<= >= < |= = > {if(!COMMENT) printf("\n\t%s is a RELATIONAL
OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file =
fopen(argv[1],"r");
if(!file)
{
printf("could not open %s
\n",argv[1]); exit(0);
}
yyin = file;
}
yylex();
printf("\n\n");
return 0;
} int yywrap()
{
return 0;
}

```

## 6.6 INPUT & OUTPUT:

### Input

```
$vi var.c  
#include<stdio.h>
```

```
main()  
{  
int a,b;  
}
```

### Output

```
$lex lex.l
```

```
$cc lex.yy.c
```

```
$/a.out var.c  
#include<stdio.h> is a PREPROCESSOR DIRECTIVE  
FUNCTION
```

```
main (  
)  
BLOCK BEGINS  
int is a KEYWORD  
a IDENTIFIER  
  
b IDENTIFIER BLOCK ENDS
```

## 6.7 LAB ASSIGNMENT:

1. Write a program that defines auxiliary definitions and translation rules of Pascal tokens?
2. Write a program that defines auxiliary definitions and translation rules of C tokens?
3. Write a program that defines auxiliary definitions and translation rules of JAVA tokens

## EXPERIMENT-7

### 7.1 OBJECTIVE:

Write a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.

### 7.2 RESOURCE:

Turbo C++, Codeblock

### 7.3 PROGRAM LOGIC:

Read the input string.

By using the FIRST AND FOLLOW values.

Verify the FIRST of non terminal and insert the production in the FIRST value

If we have any @ terms in FIRST then insert the productions in FOLLOW values

Constructing the predictive parser table

#### 7.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

#### 7.5 PROGRAM:

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

```
#include<conio.h>
```

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

```
char prol[7][10]={"S","A","A","B","B","C","C"};
```

```
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
```

```
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"}; char  
first[7][10]={"abcd","ab","cd","a@","@","c@","@"}; char  
follow[7][10]={"$","$","$","a$","b$","c$","d$"};
```

```
char table[5][6][10];
```

```
numr(char c)
```

```
{
```

```
switch(c)
```

```
{
```

```
case 'S': return 0;
```

```
case 'A': return 1;
```

```
case 'B': return 2;
```

```
case 'C': return 3;
```

```
case 'a': return 0;
```

```

case 'b': return 1;

case 'c': return 2;

case 'd': return 3;

case '$': return 4;

}
return(2);

}

void main()

{

int i,j,k;

clrscr();

for(i=0;i<5;i++)

for(j=0;j<6;j++)

strcpy(table[i][j], " ");

printf("\nThe following is the predictive parsing table for the following grammar:\n");
for(i=0;i<7;i++)

printf("%s\n",prod[i]);

printf("\nPredictive parsing table is\n");

fflush(stdin);

for(i=0;i<7;i++)

{

k=strlen(first[i]);

for(j=0;j<10;j++)

```

```

    if(first[i][j]!='@')

    strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);

    }

    for(i=0;i<7;i++)

    {

    if(strlen(pror[i])==1)

    {

    if(pror[i][0]=='@')

    {

    k=strlen(follow[i]);

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

    strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);

}

    }

    strcpy(table[0][0]," ");

    strcpy(table[0][1],"a");

    strcpy(table[0][2],"b");

    strcpy(table[0][3],"c");

    strcpy(table[0][4],"d");

    strcpy(table[0][5],"$");

    strcpy(table[1][0],"S");

    strcpy(table[2][0],"A");

    strcpy(table[3][0],"B");

    strcpy(table[4][0],"C");

    printf("\n-----\n");

```



```

for(i=0;i<5;i++)

for(j=0;j<6;j++)

{

printf("%-10s",table[i][j]);

if(j==5)

printf("\n-----\n");

}

getch();

}

```

## 7.6 INPUT & OUTPUT:

The following is the predictive parsing table for the following grammar:

S->A

A->Bb

A->Cd

B->aB

B->@

C->Cc

C->@

Predictive parsing table is

-----				
a	b	c	d	\$
-----				
S	S->AS->AS->AS->A			
-----				
A	A->Bb A->BbA->Cd A->Cd			
-----				

B B->aB B->@ B->@ B->@

C C->@C->@ C->@

### 7.7 LAB ASSIGNMENT:

1. Write a program to compute FIRST for the following grammar?

E → TE'

E' → +TE'T'FT'

T' → \*FT'F(E)/i

2. Write a program to compute FIRST for the following grammar?

S → iCtSS' S' → eS/î

## EXPERIMENT-8

### 8.1 OBJECTIVE:

Write a C program for constructing of LL (1) parsing.

### 8.2 RESOURCE:

Turbo C++, Codeblock

### 8.3 PROGRAM LOGIC:

Read the input string. Using predictive parsing table parse the given input using stack .

If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to \$.

### 8.4 PROCEDURE:

Go to debug -> run or press CTRL + F9 to run the program.

### 8.5 PROGRAM

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

```
#include<conio.h>
```

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

```
char s[20],stack[20];
```

```
void main()
```

```
{
```

```
char m[5][6][3]={ "tb"," ","","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," ","",
"n","*fc"," " a "n","n","i"," "," ","(e)"," "," "};
```

```
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
```

```
int i,j,k,n,str1,str2;
```

```
clrscr();
```

```
printf("\n Enter the input string: ");
```

```
scanf("%s",s);
```

```
strcat(s,"$");
```

```
n=strlen(s);
```

```
stack[0]='$';
```

```
stack[1]='e';
```

```
i=1;
```

j=0;

```
printf("\nStack    Input\n");
```

```
printf("_____\n");
```

```
while((stack[i]!='$')&&(s[j]!='$'))
```

$$\{$$

```
if(stack[i]==s[j])
```

$$\{$$
 $\dot{\mathbf{i}}_{--};$ 

j++;

$$\}$$

```
switch(stack[i])
```

 $\{$

```
case 'e': str1=0;
```

```
break;
```

```
case 'b': str1=1;
```

```
break;
```

```
case 't': str1=2;
```

```
break;
```

```
case 'c': str1=3;
```

```
break;
```

```
case 'f': str1=4;
```

```
break;
```

```
}
```

```
switch(s[j])
```

```
{
```

```
case 'i': str2=0;
```

```
break;
```

```
case '+': str2=1;
```

```
break;
```

```
case '*': str2=2;
```

```
break;
```

```
case '(': str2=3;
```

```
break;
```

```
case ')': str2=4;
```

```
break;
```

```
case '$': str2=5;
```

```
break;
```

```

    }

    if(m[str1][str2][0]=="\0")

    {

        printf("\nERROR");

        exit(0);

    }

    else if(m[str1][str2][0]=='n')

        i--;

    else if(m[str1][str2][0]=='i')
        stack[i]='i';

    else

    {

        for(k=size[str1][str2]-1;k>=0;k--)

        {

            stack[i]=m[str1][str2][k];

            i++;

        }

        i--;

    }

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

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

    printf("    ");

    for(k=j;k<=n;k++)

```

```

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

printf(" \n ");

}

printf("\n SUCCESS");

getch(); }

```

## **EXPERIMENT-9**

### **9.1 OBJECTIVE:**

Write a C program to calculate FIRST of a regular expression.

### **9.2 RESOURCE:**

Turbo C++, Codeblock

### **9.3 PROGRAM LOGIC:**

To compute FIRST(X) for all grammar symbols x, apply the following rules until no more

terminals can be added to any FIRST set.

1. if X is terminal, then FIRST(X) is {X}.
2. if X is nonterminal and  $X \rightarrow a\alpha$  is a production, then add a to FIRST(X). if  $X \rightarrow \epsilon$  to FIRST(X)
3. if  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production, then for all i such that all of  $Y_1, \dots, Y_{i-1}$  are nonterminals and FIRST( $Y_j$ ) contains  $\epsilon$  for  $j=1, 2, \dots, i-1$ , add every non- $\epsilon$  symbol in FIRST( $Y_i$ ) to FIRST(X). if V is in FIRST( $Y_j$ ) for  $j=1, 2, \dots, k$ , then add  $\epsilon$  to FIRST(X).

### **9.4 PROGRAM**

```

#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char t[5],nt[10],p[5][5],first[5][5],temp;
int i,j,not,nont,k=0,f=0;
clrscr();
printf("\nEnter the no. of Non-terminals in the grammer:");
scanf("%d",&nont);
printf("\nEnter the Non-terminals in the grammer:\n");
for(i=0;i<nont;i++)

```

```

{
scanf("\n%c",&nt[i]);
}
printf("\nEnter the no. of Terminals in the grammer: ( Enter e for absiline ) ");
scanf("%d",&not);
printf("\nEnter the Terminals in the grammer:\n");
for(i=0;i<not||t[i]!='$';i++)
{
scanf("\n%c",&t[i]);
}
for(i=0;i<nont;i++)
{
p[i][0]=nt[i];
first[i][0]=nt[i];
}
printf("\nEnter the productions :\n");
for(i=0;i<nont;i++)
{
scanf("%c",&temp);
printf("\nEnter the production for %c ( End the production with '$' sign )
:",p[i][0]);
for(j=0;p[i][j]!='$';)
{
j+=1;
scanf("%c",&p[i][j]);
}
}
for(i=0;i<nont;i++)
{
printf("\nThe production for %c -> ",p[i][0]);
for(j=1;p[i][j]!='$';j++)
{
printf("%c",p[i][j]);
}
}
for(i=0;i<nont;i++)

{
f=0;
for(j=1;p[i][j]!='$';j++)
{
for(k=0;k<not;k++)
{
if(f==1)
break;
if(p[i][j]==t[k])
{
first[i][j]=t[k];
first[i][j+1]='$';

```

```

f=1;

break;
}
else if(p[i][j]==nt[k])
{
first[i][j]=first[k][j];
if(first[i][j]=='e')
continue;
first[i][j+1]='$';
f=1;
break;
}
}
}
}
for(i=0;i<nont;i++)
{
printf("\n\nThe first of %c -> ",first[i][0]);
for(j=1;first[i][j]!='$';j++)
{
printf("%c\t",first[i][j]);
}
}
getch();
}

```

## 9.5 INPUT/OUTPUT

Enter the no. of Non-terminals in the grammer:3

Enter the Non-terminals in the grammer:

ERT

Enter the no. of Terminals in the grammer: ( Enter e for absiline ) 5

Enter the Terminals in the grammer:

ase\*+

Enter the productions :

Enter the production for E ( End the production with '\$' sign ) :a+s\$

Enter the production for R ( End the production with '\$' sign ) :e\$

Enter the production for T ( End the production with '\$' sign ) :Rs\$

The production for E -> a+s

The production for R -> e

The production for T -> Rs

The first of E -> a

The first of R -> e

The first of T -> e s



## EXPERIMENT-10

### TO CALCULATE LEADING OF NON-TERMINALS

#### ALGORITHM

1. Start
2. For each nonterminal A and terminal a **do** L(A,a):= **false**;
3. For each production of the form A->a or A->B **do**

INSTALL(A,a);

4. **While** **STACK** not empty **repeat** step 5& 6

5. Pop top pair (B,a) from **STACK**;

6. For each production of the form A->B **do**

INSTALL(A,a)

7. Stop

#### Algorithm For INSTALL(A,a)

1. Start
2. If L(A,a) not present do step 3 and 4.
- 3 . Make L(A,a)=True
- 4 . Push (A,a) onto stack
- 5 . Stop

#### PROGRAM IS TO CALCULATE LEADING FOR ALL THE NON-TERMINALS OF THE GIVEN GRAMMAR

```
#include<conio.h>
```

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

```
char arr[18][3] =
```

```
{  
{ 'E','+', 'F'}, { 'E','*', 'F'}, { 'E','(', 'F'}, { 'E',')', 'F'}, { 'E','i', 'F'}, { 'E','$', 'F'},  
{ 'F','+', 'F'}, { 'F','*', 'F'}, { 'F','(', 'F'}, { 'F',')', 'F'}, { 'F','i', 'F'}, { 'F','$', 'F'},  
{ 'T','+', 'F'}, { 'T','*', 'F'}, { 'T','(', 'F'}, { 'T',')', 'F'}, { 'T','i', 'F'}, { 'T','$', 'F'},  
};
```

```
char prod[6] = "EETTFF";
```

```
char res[6][3]=
```

```
{  
{ 'E','+', 'T'}, { 'T','\0'},  
{ 'T','*', 'F'}, { 'F','\0'},  
{ '(', 'E',')'}, { 'i','\0'},  
};
```

```
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;
}
}
++top;
stack[top][0]=pro;
stack[top][1]=re;
}
void main()
{
int i=0,j;
char pro,re,pri=' ';
clrscr();
for(i=0;i<6;++i)
{
for(j=0;j<3 && res[i][j]!='\0';++j)
{
if(res[i][j]
=='+'||res[i][j]=='*'||res[i][j]=='('||res[i][j]==')'||res[i][j]=='i'||res[i][j]=='$')
{
install(prod[i],res[i][j]);
break;
}
}
}
while(top>=0)
{
pro = stack[top][0];
re = stack[top][1];
--top;
for(i=0;i<6;++i)
{
if(res[i][0]==pro && res[i][0]!=prod[i])
{
install(prod[i],re);
}
}
}
for(i=0;i<18;++i)
{
printf("\n\t");
for(j=0;j<3;++j)
printf("%c\t",arr[i][j]);
}
getch();
clrscr();
printf("\n\n");

```

```

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]);
}
getch();}

```

### OUTPUT

```

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

```

## EXPERIMENT-11

### Definition

JFLAP defines a finite automaton (FA)  $M$  as the quintuple  $M = (Q, \Sigma, \delta, q_S, F)$  where

$Q$  is a finite set of states  $\{q_i \mid i \text{ is a nonnegative integer}\}$   $\Sigma$  is the finite input alphabet

$\delta$  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

Note that this definition includes both deterministic finite automata (DFAs), which we will

be discussing shortly, and nondeterministic finite automata (NFAs), which we will touch on later.

Building the different types of automata in JFLAP is fairly similar, so let's start by building a DFA for the language  $L =$

$\{a^m b^n : m \geq 0, n > 0, n \text{ is odd}\}$ . That is, we will build a DFA that recognizes that language of any number of  $a$ 's followed by any odd number of  $b$ 's. (Examples taken from *JFLAP: An Interactive Formal Languages and Automata Package* by Susan Rodger and Thomas Finley.)

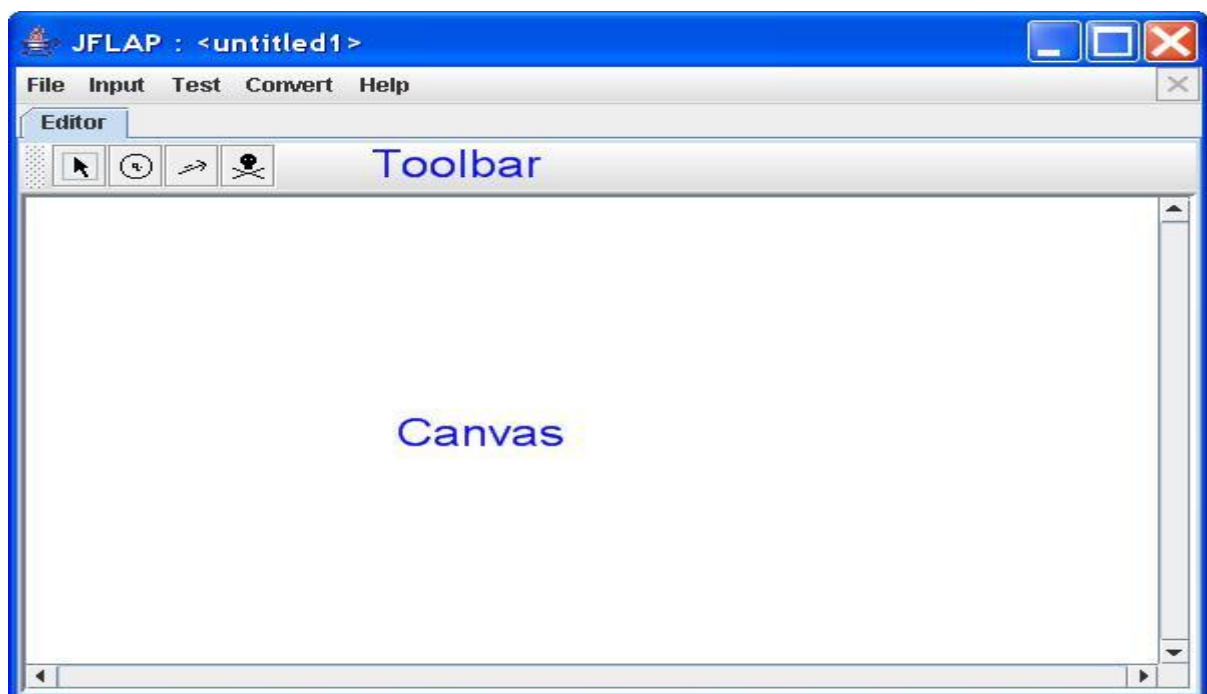
## The Editor Window

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



Starting a new FA

This should bring up a new window that allows you to create and edit an FA. The editor is divided into two basic areas: the canvas, which you can construct your automaton on, and the toolbar, which holds the tools you need to construct your automaton.



The editor window

Let's take a closer look at the toolbar.




The FA toolbar


As you can see, the toolbar holds four tools:

Attribute Editor tool  : [sets initial and final states](#)

State Creator tool  : [creates states](#)

Transition Creator tool  : [creates transitions](#)


Deletor tool  : [deletes states and transitions](#)

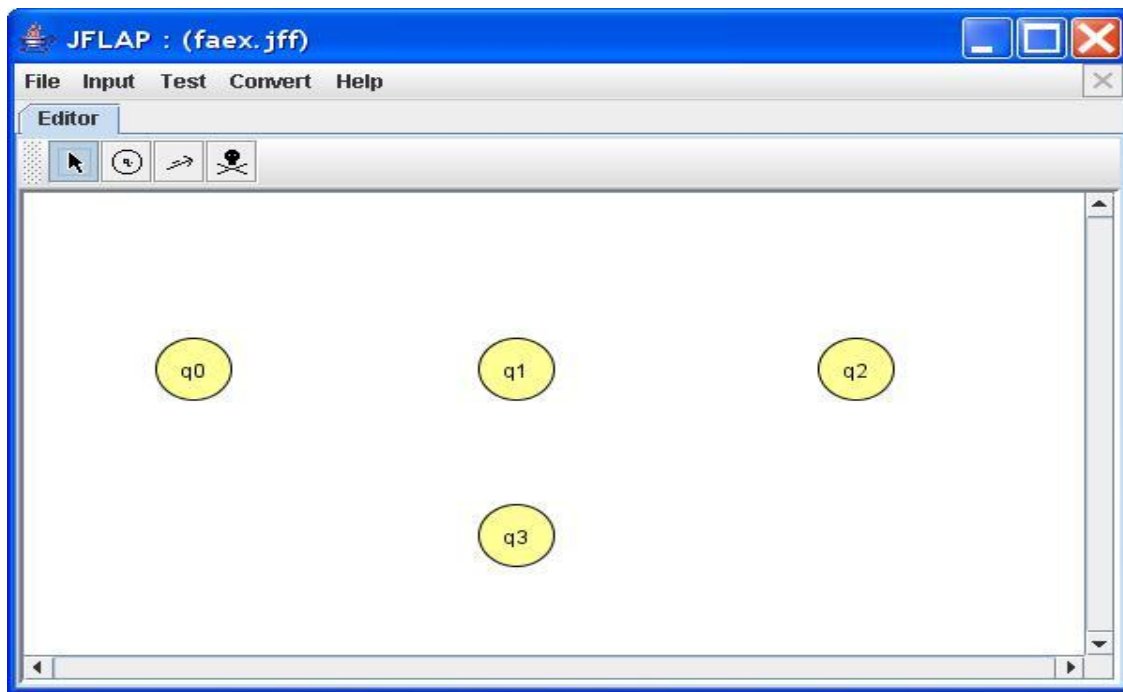
To select a tool, click on the corresponding icon with your mouse. When a tool is selected, it is shaded, as the Attribute Editor tool  is above. Selecting the tool puts you in the corresponding mode. For instance, with the toolbar above, we are now in the Attribute Editor mode.

The different modes dictate the way mouse clicks affect the machine. For example, if we are in the State Creator mode, clicking on the canvas will create new states. These modes will be described in more detail shortly.

Now let's start creating our FA.

## Creating States


First, let's create several states. To do so we need to activate that State Creator tool by clicking the  button on the [toolbar](#). Next, click on the canvas in different locations to create states. We are not very sure how many states we will need, so we created four states. Your editor window should look something like this:

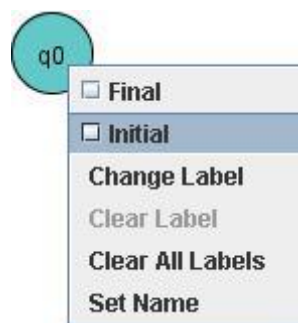


States created

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

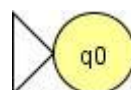
## Defining Initial and Final States

Arbitrarily, we decide that  $q_0$  will be our initial state. To define it to be our initial state, first select the Attribute Editor tool  on the [toolbar](#). Now that we are in Attribute Editor mode, right-click on  $q_0$ . This should give us a pop-up menu that looks like this:



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 initial state.



$q_0$  defined as initial state

Next, let's create a final state. Arbitrarily, we select  $q_1$  as our 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.




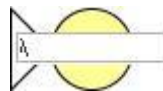
$q_1$  defined as final state

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

## Creating Transitions

We know strings in our [language](#) can start with  $a$ 's, so, the initial state must have an outgoing transition on  $a$ . We also know that it can start with any number of  $a$ 's, which means that the FA should be in the same state after processing input of any number of  $a$ 's. Thus, the outgoing transition on  $a$  from  $q_0$  loops back to itself.

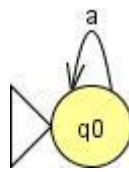
To create such a transition, first select the Transition Creator tool  from the [toolbar](#). Next, click on  $q_0$  on the canvas. A text box should appear over the state:




Creating a transition

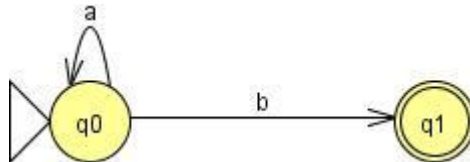
Note that  $\lambda$ , representing the empty string, is initially filled in for you. If you prefer  $\epsilon$  representing the empty string, select Preferences : Preferences in the [main menu](#) to change the symbol representing the empty string.

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:



Transition created

Next, we know that strings in our [language](#) must end with a odd number of  $b$ 's. Thus, we know that the outgoing transition on  $b$  from  $q_0$  must be to a final state, as a string ending with one  $b$  should be accepted. To create a transition from our 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 same way you entered "a" for the previous transition. The transition between two states should look like this:

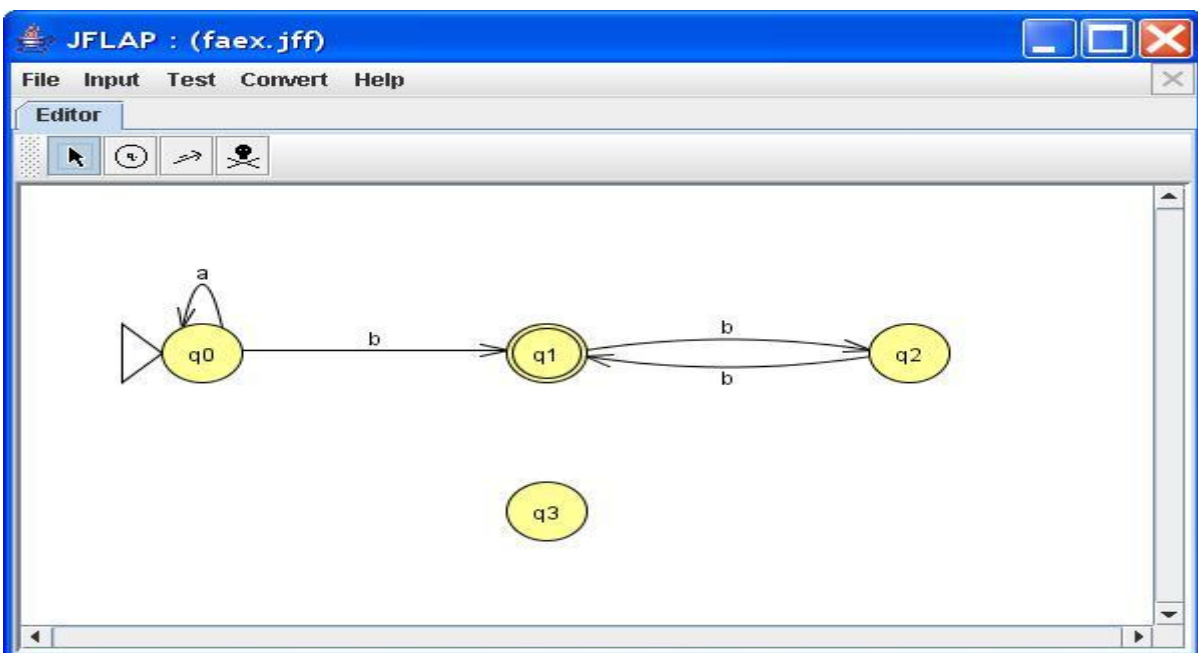


Second transition created

Lastly, we know that only strings that end with an odd number of  $b$ 's should be accepted. Thus, we know that  $q_1$  has an outgoing transition on  $b$ , that it cannot loop back to  $q_1$ . There are two options for the transition: it can either go to the initial state  $q_0$ , or to a brand new state, say,  $q_2$ .

If the transition on  $b$  was to the initial state  $q_0$ , strings would not have to be of the form  $a^m b^n$ ; strings such as  $ababab$  would also be accepted. Thus, the transition cannot be to  $q_0$ , and it must be to  $q_2$ .

Create a transition on  $b$  from  $q_1$  to  $q_2$ . As the FA should accept strings that end with an odd number of  $b$ 's, create another transition on  $b$  from  $q_2$  to  $q_1$ . Your FA is now a full, working FA! It should look something like this:




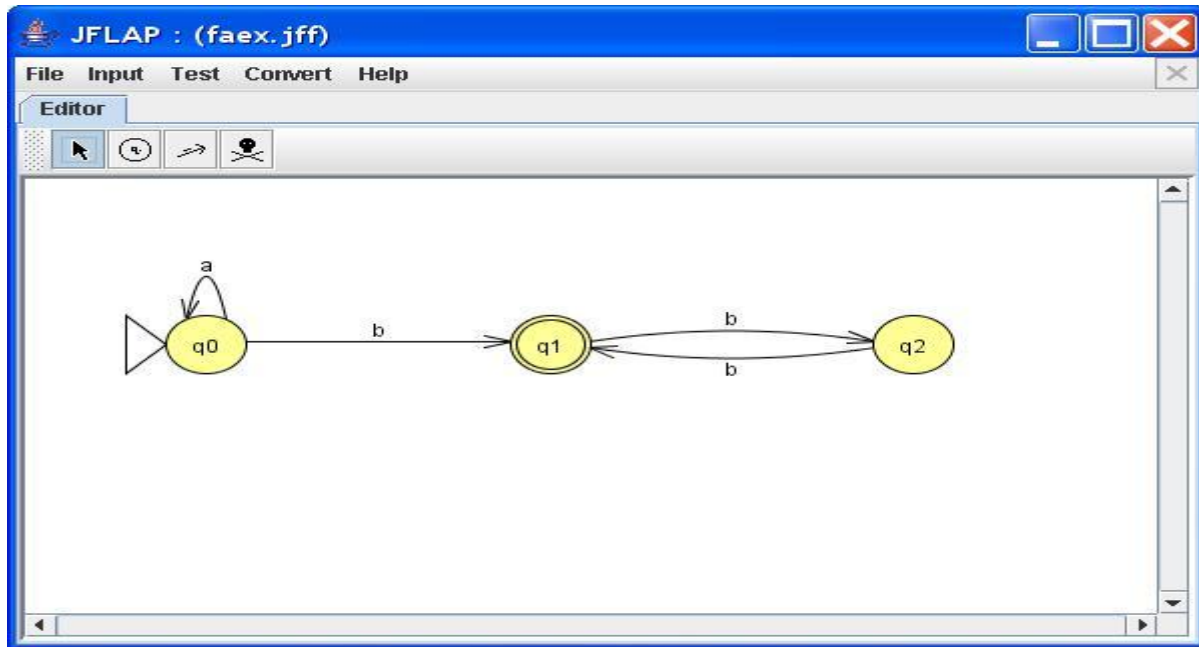


The working FA

You might notice that the  $q_3$  is not used and can be deleted. Next, we will describe how to delete states and transitions.

## Deleting States and Transitions

To delete  $q_3$ , first select the Deletor tool  on the [toolbar](#). Next, click on the state  $q_3$ . Your editor window should now look something like this:



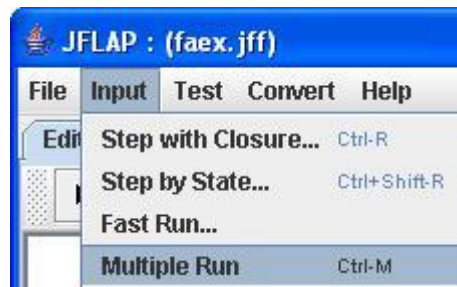
$q_3$  deleted

Similarly, to delete a transition, simply click on the input symbol of the transition when in Deletor mode.

Your FA, [faex.jff](#), is now complete.

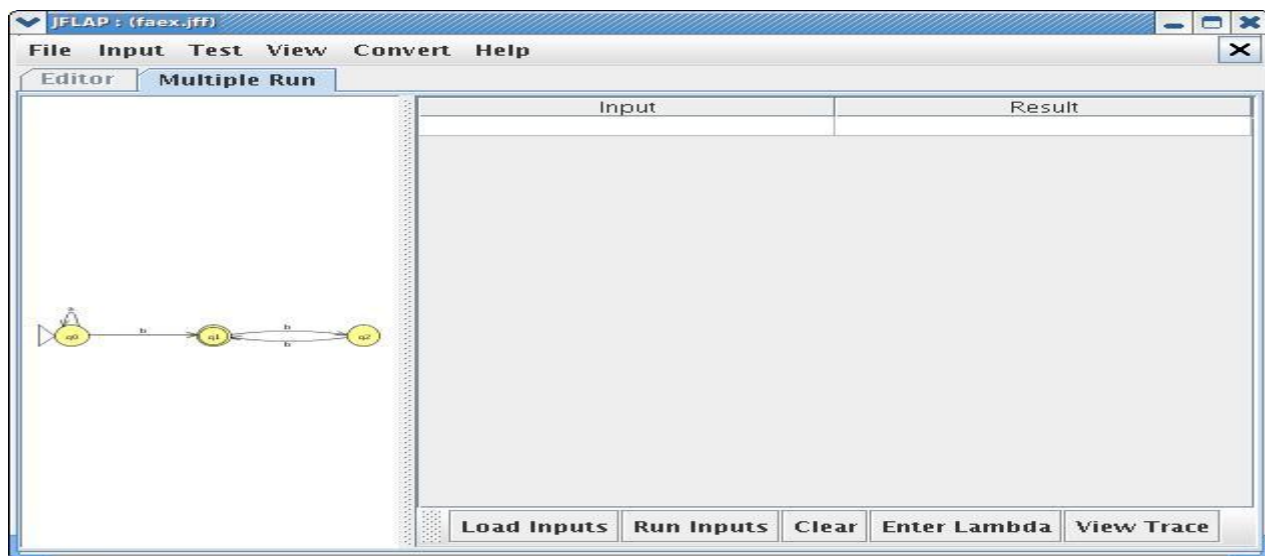
## Running the FA on Multiple Strings

Now that you've completed your FA, you might want to test it to see if it really accepts strings from the [language](#). To do so, select Input : Multiple Run from the menu bar.



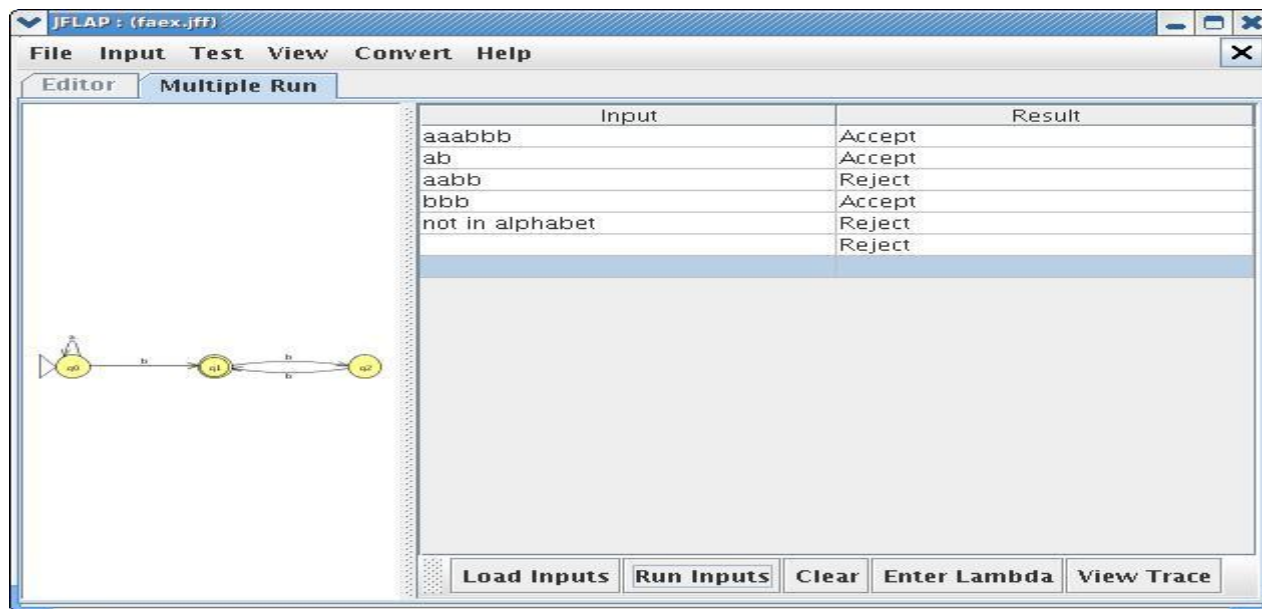
Starting a multiple run tab

A new tab will appear displaying the automaton on the left pane, and an input table on the right:



## A new multiple run tab

To enter the input strings, click on the first row in the Input column and type in the string. Press Enter to continue to the next input string. When you are done, click Run Inputs to test your FA on all the input strings. The results, Accept or Reject are displayed in the Result column. You can also load the inputs from file delimited by white space. Simply click on Load Inputs and load the file to add additional input strings into multi-run pane



## Running multiple inputs

Clicking Clear deletes all the input strings, while Enter Lambda enters the empty string at the cursor. View

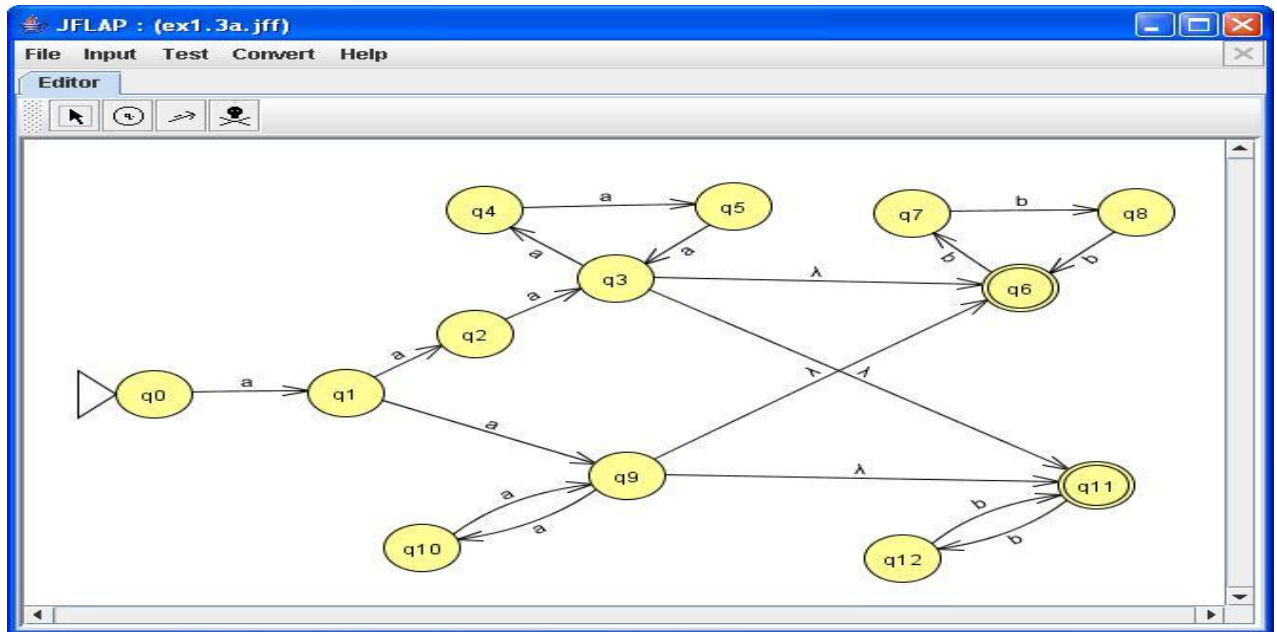
Trace brings up a separate window that shows the trace of the selected input. To return to the Editor window, select File : Dismiss Tab from the menu bar.

## Building a Nondeterministic Finite Automaton

Building a nondeterministic finite automaton (NFA) is very much like building a DFA. However, an NFA is different from a DFA in that it satisfies one of two conditions. Firstly, if the FA has two transitions from the same state that read the same symbol, the FA is considered an NFA. Secondly, if the FA has any transitions that read the empty string for input, it is also considered an NFA.

Let's take a look at this NFA, which can be accessed through [ex1.3a.jff](#): (Note: This example taken from *JFLAP: An Interactive Formal Languages and Automata Package* by Susan Rodger and Thomas Finley.)

.

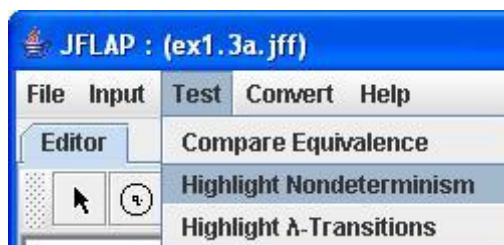


An NFA

We can immediately tell that this is an NFA because of the four  $\lambda$ -transitions coming from  $q_3$  and  $q_9$ , but we might not be sure if we have spotted all the nondeterministic states. JFLAP can help with that.

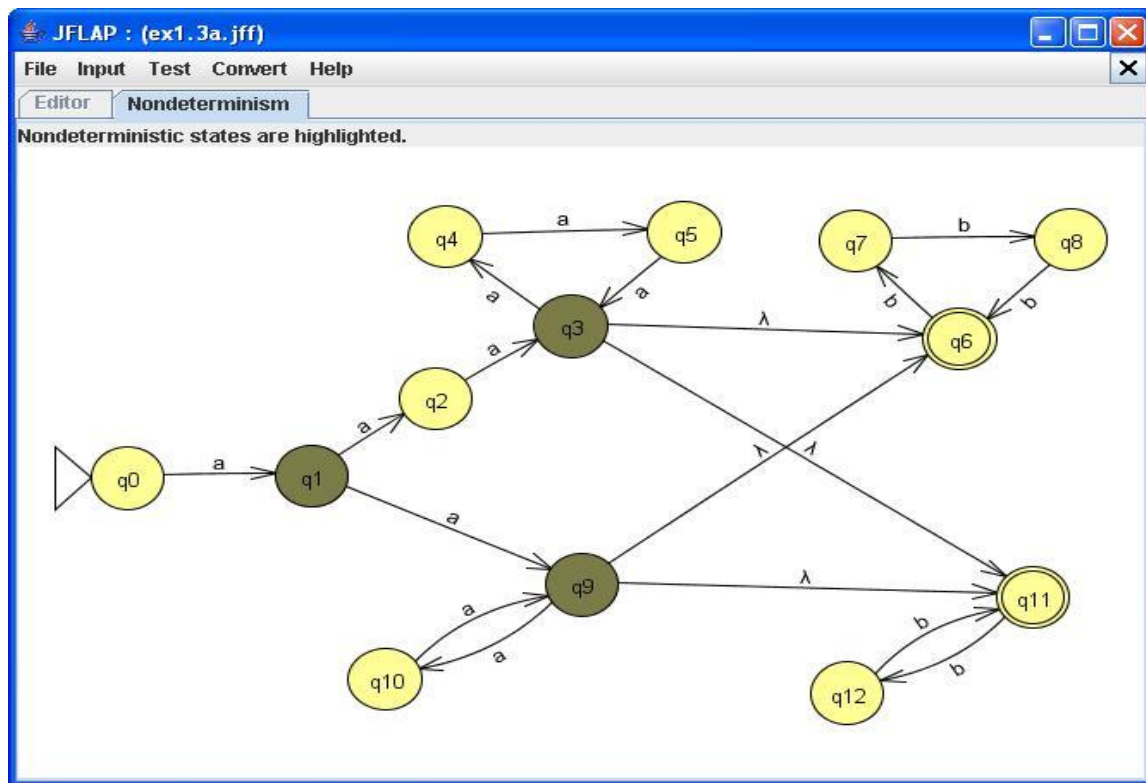
## Highlighting Nondeterministic States

To see all the nondeterministic states in the NFA, select Test : Highlight Nondeterminism from the menu bar:



Highlighting nondeterministic states

A new tab will appear with the nondeterministic states shaded a darker color:



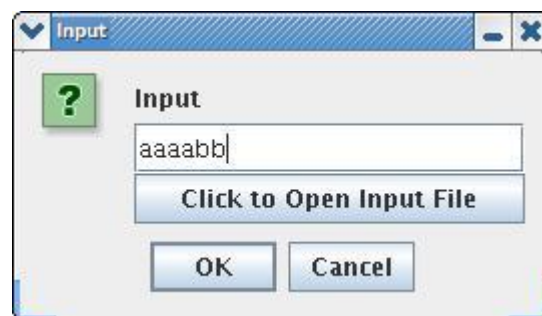
Nondeterministic states highlighted

As we can see,  $q_3$  and  $q_9$  are indeed nondeterministic because of their outgoing  $\lambda$ -transitions. Note that they would both be nondeterministic even if they each had one  $\lambda$ -transition instead of two: only one  $\lambda$ -transition is needed to make a state nondeterministic. We also see that  $q_1$  is nondeterministic because two of its outgoing transitions are on the same symbol,  $a$ . Next, we will go through JFLAP's tools for running input on an NFA.

## Running Input on an NFA

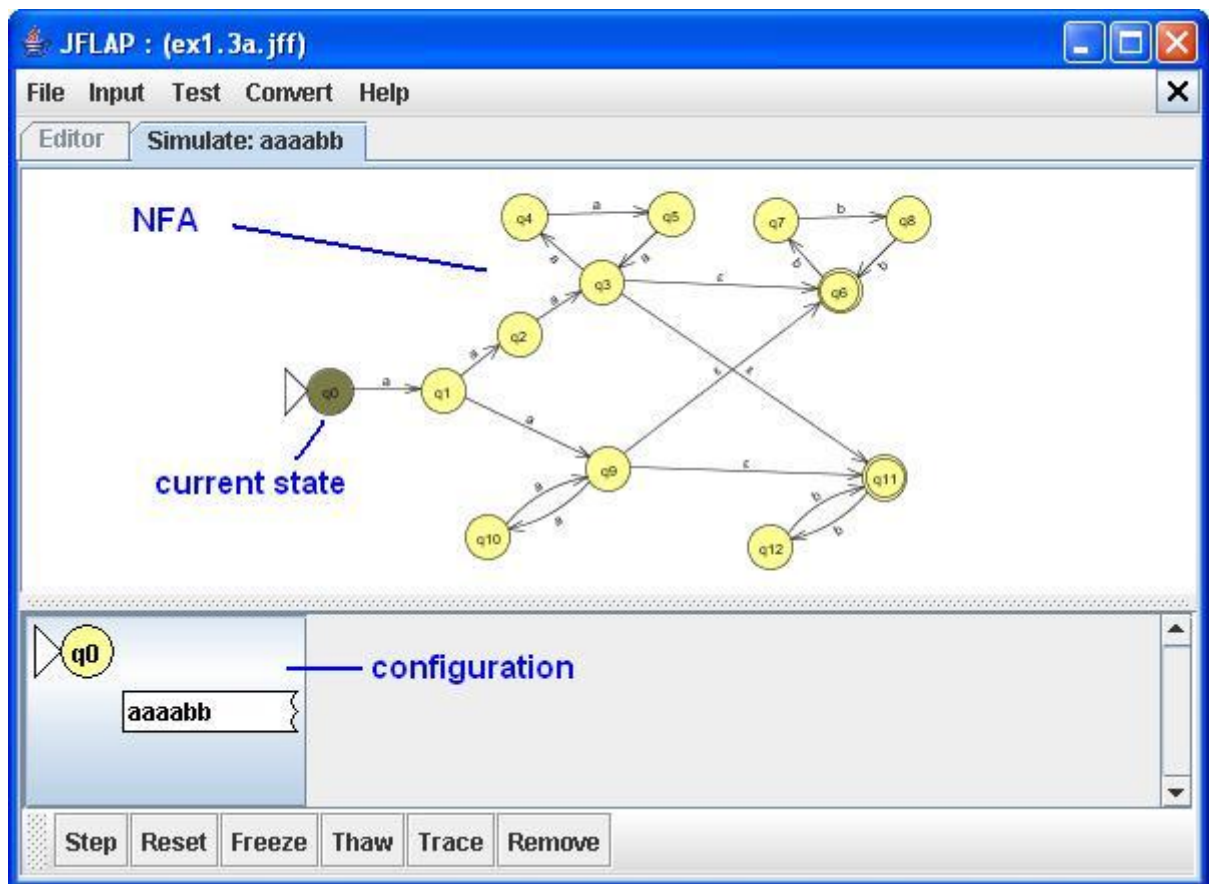
To step through input on an NFA, select Input : Step with Closure... from the menu bar. A dialog box prompting you for input will appear. Ordinarily, you would enter the input you wish to step through here. For now, type "aaaabb" in the dialog box and press Enter. You can also load the input file instead of typing the string.

NOTE : When loading input from the file, JFLAP determines end of input string by the white space. Thus if there is string "ab cd" in a file, only "ab" will be considered as an input ("cd" will be ignored since there is a white space before them).



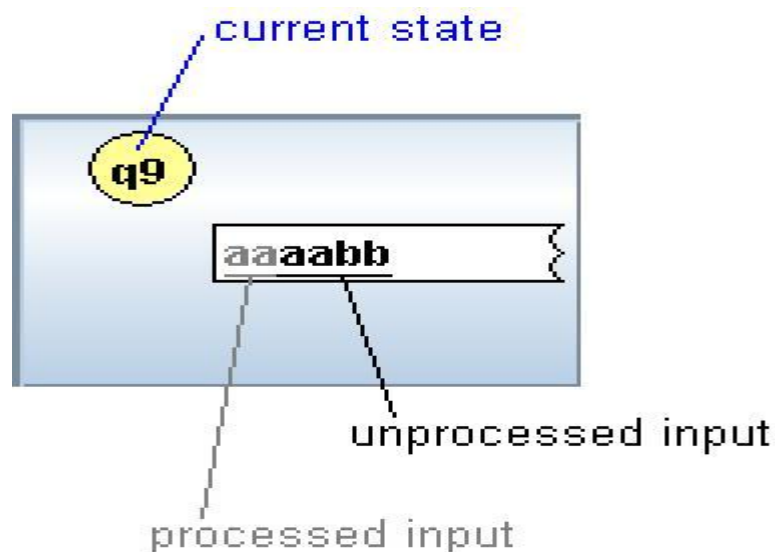
Entering input

A new tab will appear displaying the automaton at the top of the window, and configurations at the bottom. The current state is shaded.



A new input tab

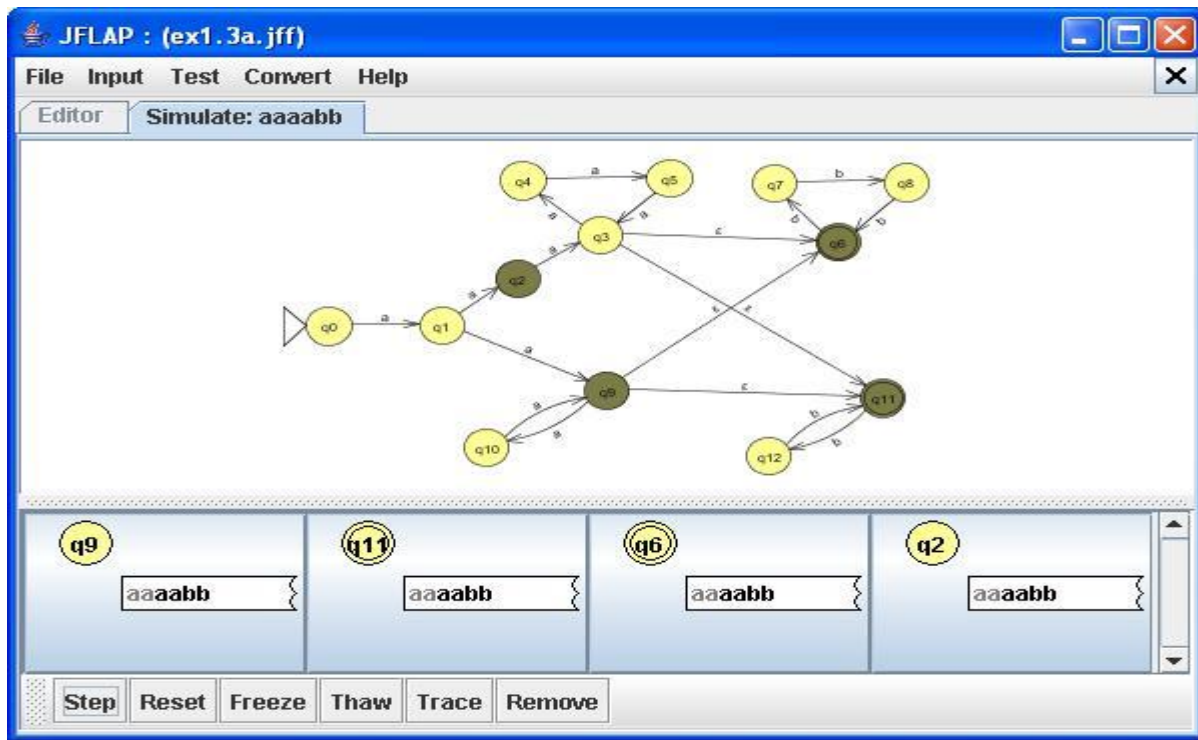
First, let's take a closer look at a configuration:



## A configuration icon

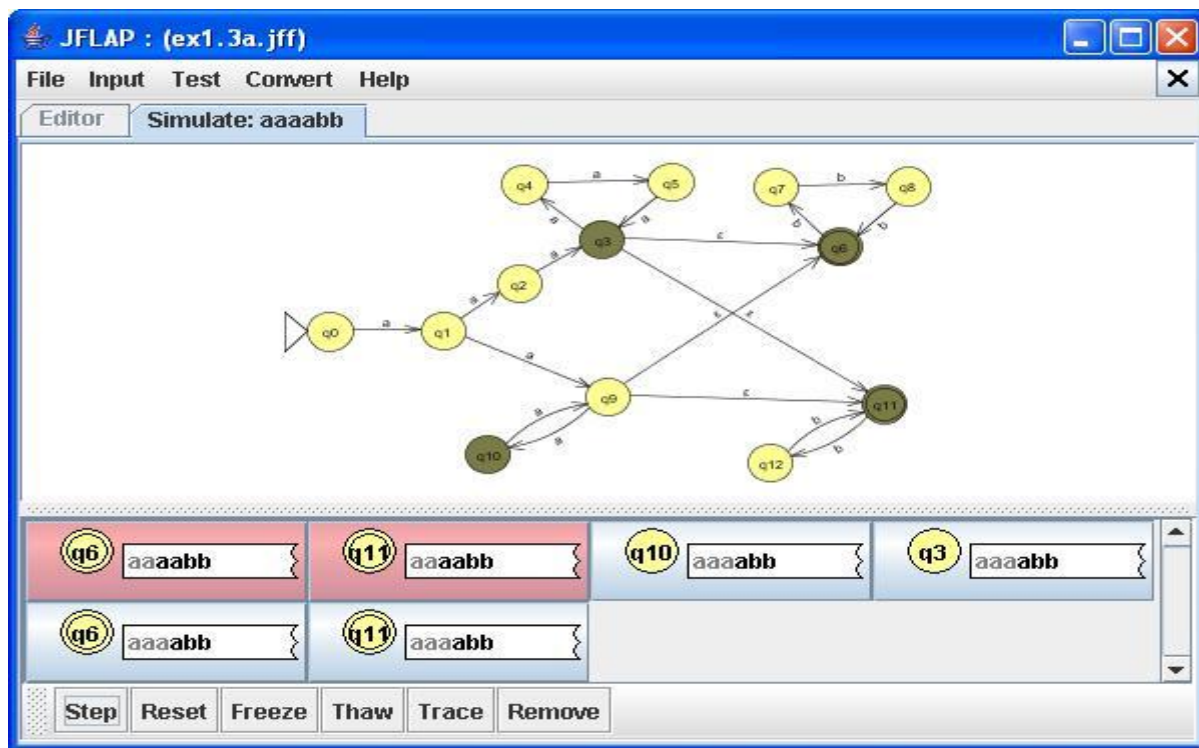
The configuration icon shows the current state of the configuration in the top left hand corner, and input on the white tape below. The processed input is displayed in gray, and the unprocessed input is black.

Click Step to process the next symbol of input. You will notice  $q_1$  becomes the shaded state in the NFA, and that the configuration icon changes, reflecting the fact that the first  $a$  has been processed. Click Step again to process the next  $a$ . You will find that four states are shaded instead of one, and there are four configurations instead of one.



*aa* processed

This is because the machine is nondeterministic. From  $q_1$ , the NFA took both *a* transitions to  $q_2$  and  $q_9$ . As  $q_9$  has two  $\lambda$ -transitions (which do not need input), the NFA further produced two more configurations by taking those transitions. Thus, the simulator now has four configurations. Click Step again to process the next input symbol.



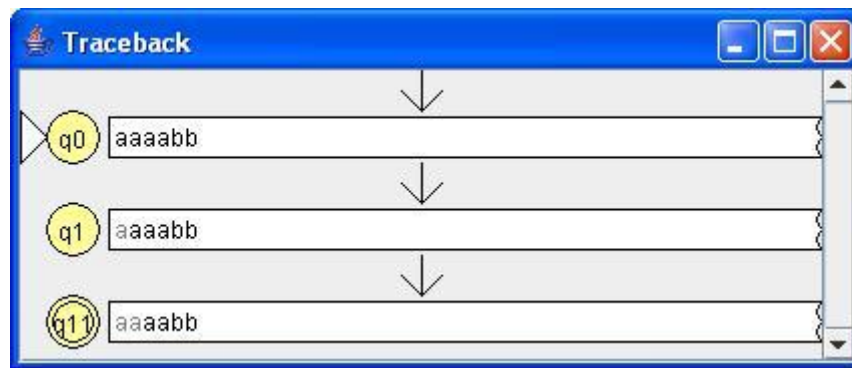


*aaa* processed

Notice that two of the configurations are highlighted red, indicating they were rejected. Looking at their input, we also know that only *aa* was processed. What happened?

## Producing a Trace

To select a configuration, click on it. It will become a solid color when selected, instead of the slightly graded color. Click on the icon for the rejected configuration with state  $q_{11}$ , and click Trace. A new window will appear showing the traceback of that configuration:



### A configuration's traceback

The traceback shows the configuration after processing each input symbol. From the traceback, we can tell that that configuration started at  $q_0$  and took the transition to  $q_1$  after processing the first *a*. After processing the second *a*, it was in  $q_{11}$ . Although  $q_{11}$  is not adjacent to  $q_1$ , it can be reached by taking a  $\lambda$ -transition from  $q_9$ . As the simulator tried to process the next *a* on this configuration, it realized that there are no outgoing *a* transitions from  $q_{11}$  and thus rejected the configuration.

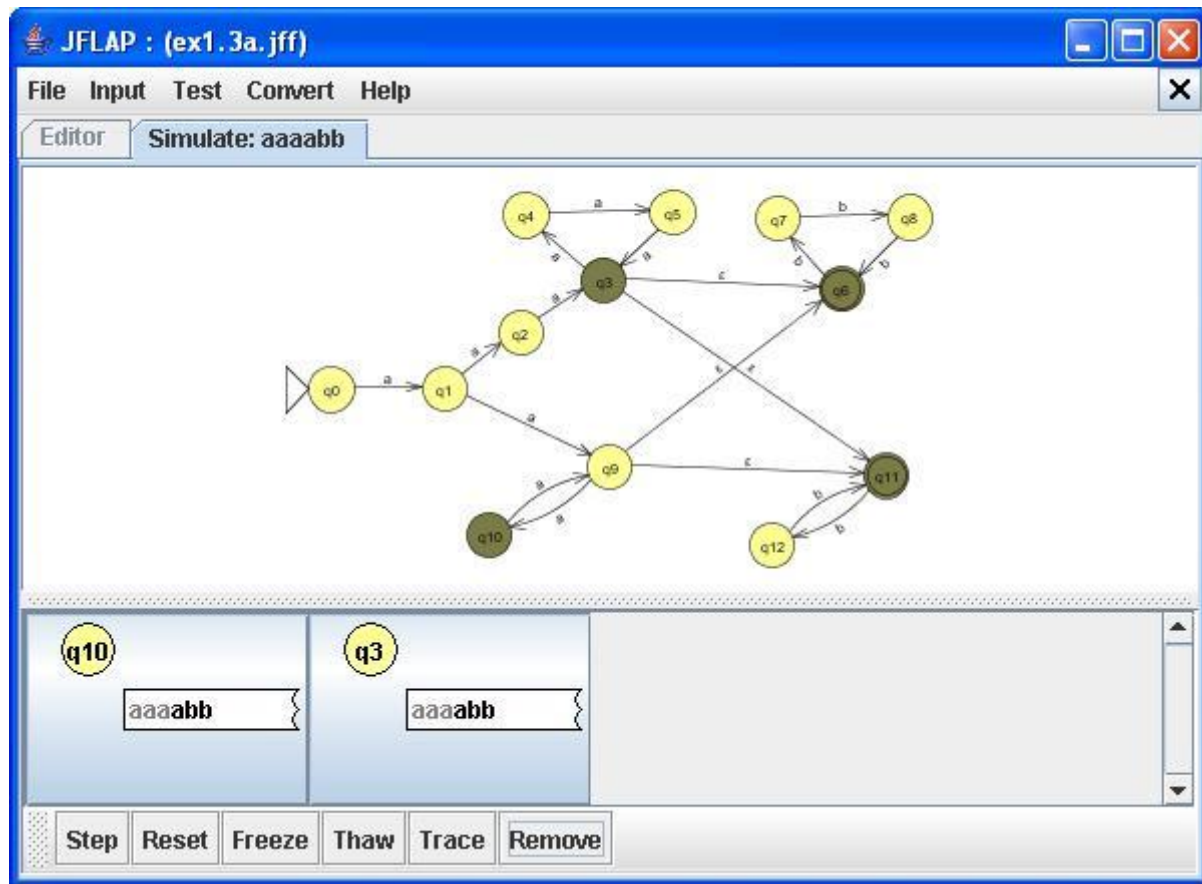
Although rejected configurations will remove themselves in the next step, we can also remove configurations that have not been rejected.

## Removing Configurations

Looking at the tracebacks of the rejected configurations, we can tell that any configurations that are in  $q_{11}$  or  $q_6$  and whose next input symbol is *a* will be rejected.

As the next input symbol is *a*, we can tell that the configurations that are currently in  $q_6$  and  $q_{11}$  will be rejected. Click once on each of the four configurations to select them, then click Remove. The simulator will no longer step these configurations. (Although we are only removing configurations that are about to be rejected, we can remove any configurations for any purpose, and the simulator will stop stepping through input on those configurations.)

Your simulator should now look something like this:



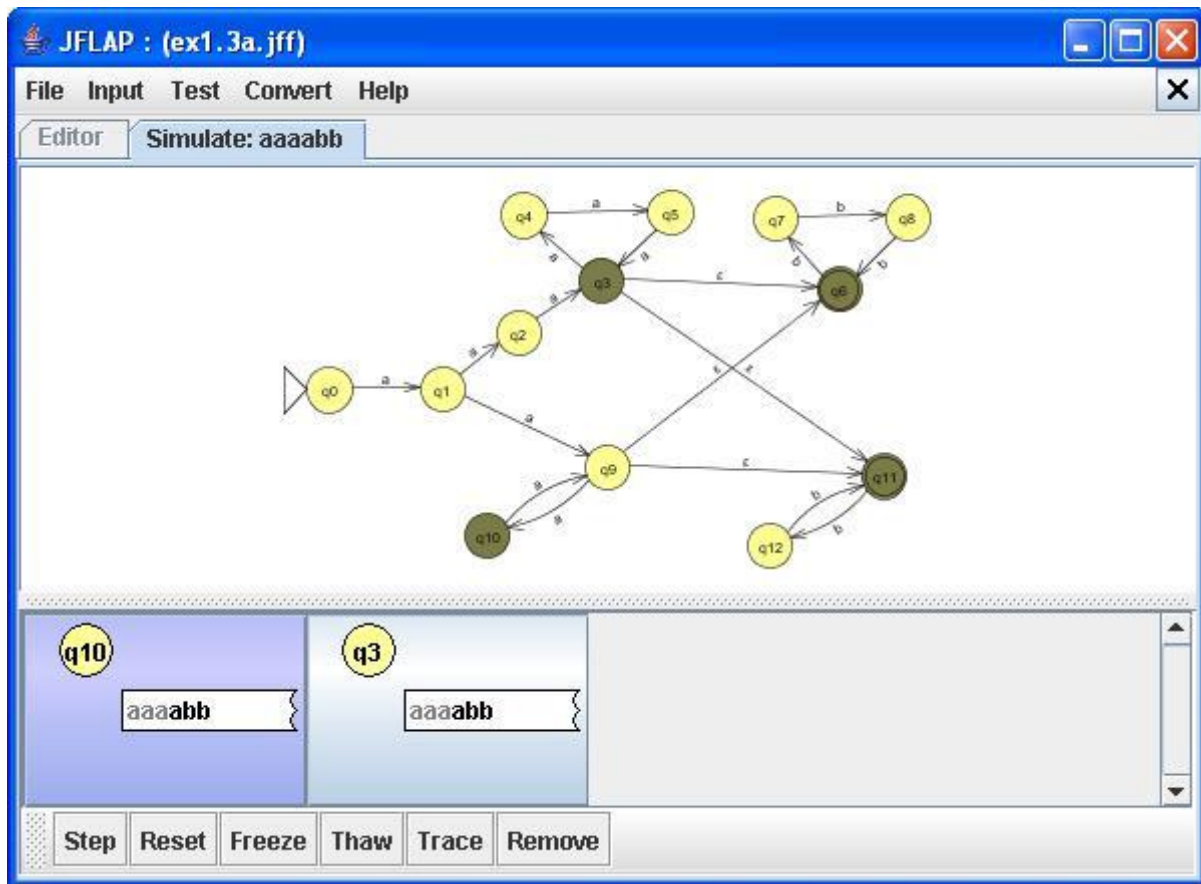
Rejected configurations removed

Now when we step the simulator, the two configurations will be stepped through.

Looking at the two configurations above, we might realize that the configuration on  $q_3$  will not lead to an accepting configuration. We can test our idea out by freezing the other configuration.

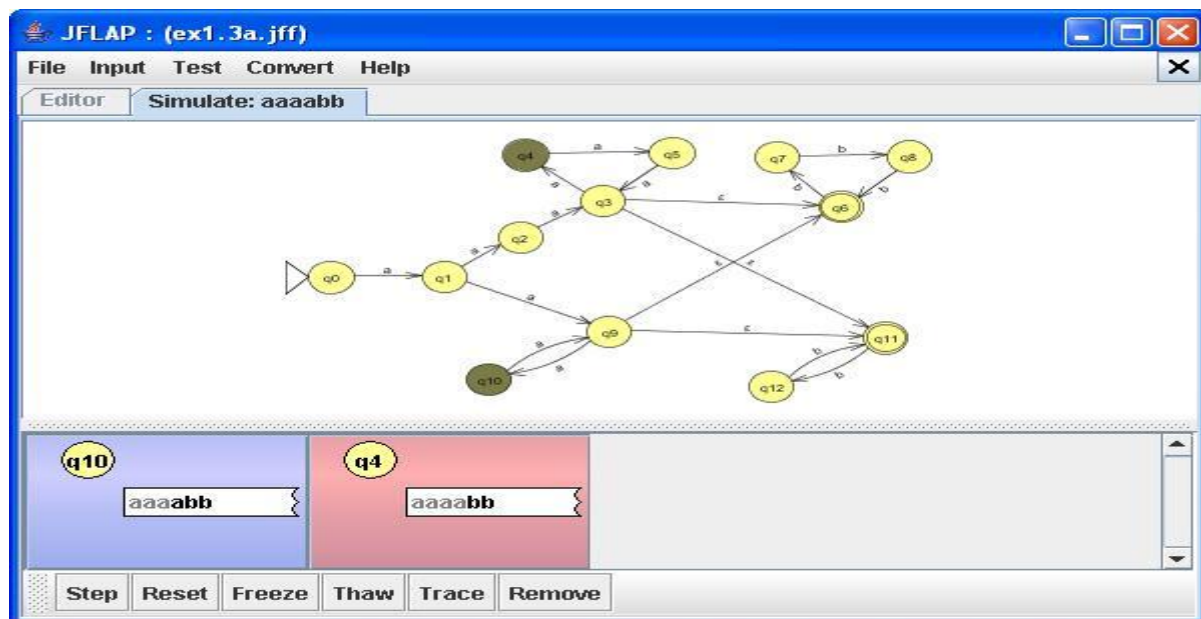
**Freezing and Thawing configurations**

To freeze the configuration on  $q_{10}$ , click on  $q_{10}$  once, then click the Freeze button. When a configuration is frozen, it will be tinted a darker shade of purple:



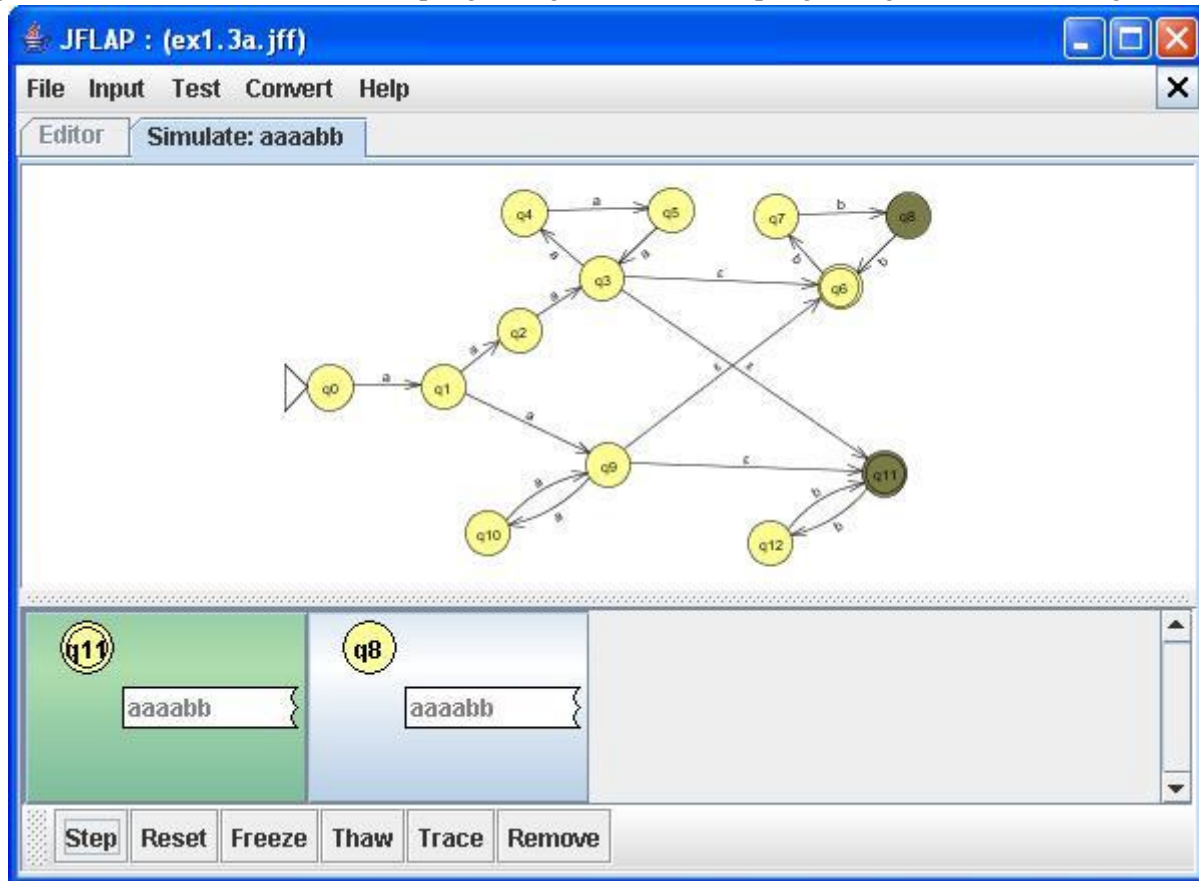
Configuration on  $q_{10}$  frozen

With that configuration frozen, as you click Step to step through the configuration on  $q_3$ , the frozen configuration remains the same. Clicking Step two more times will reveal that the configuration on  $q_3$  is not accepted either. Your simulator will now look like this:



Other configurations rejected

To proceed with the frozen configuration, select it and click Thaw. The simulator will now step through input as usual. Click Step another three times to find an accepting configuration. An accepting configuration is colored green:



Accepting configuration found

If we click Step again, we will see that the last configuration is rejected. Thus, there is only one accepting configuration. However, we might be unsure that this is really the case, as we had removed some configurations. We can double-check by resetting the simulator.

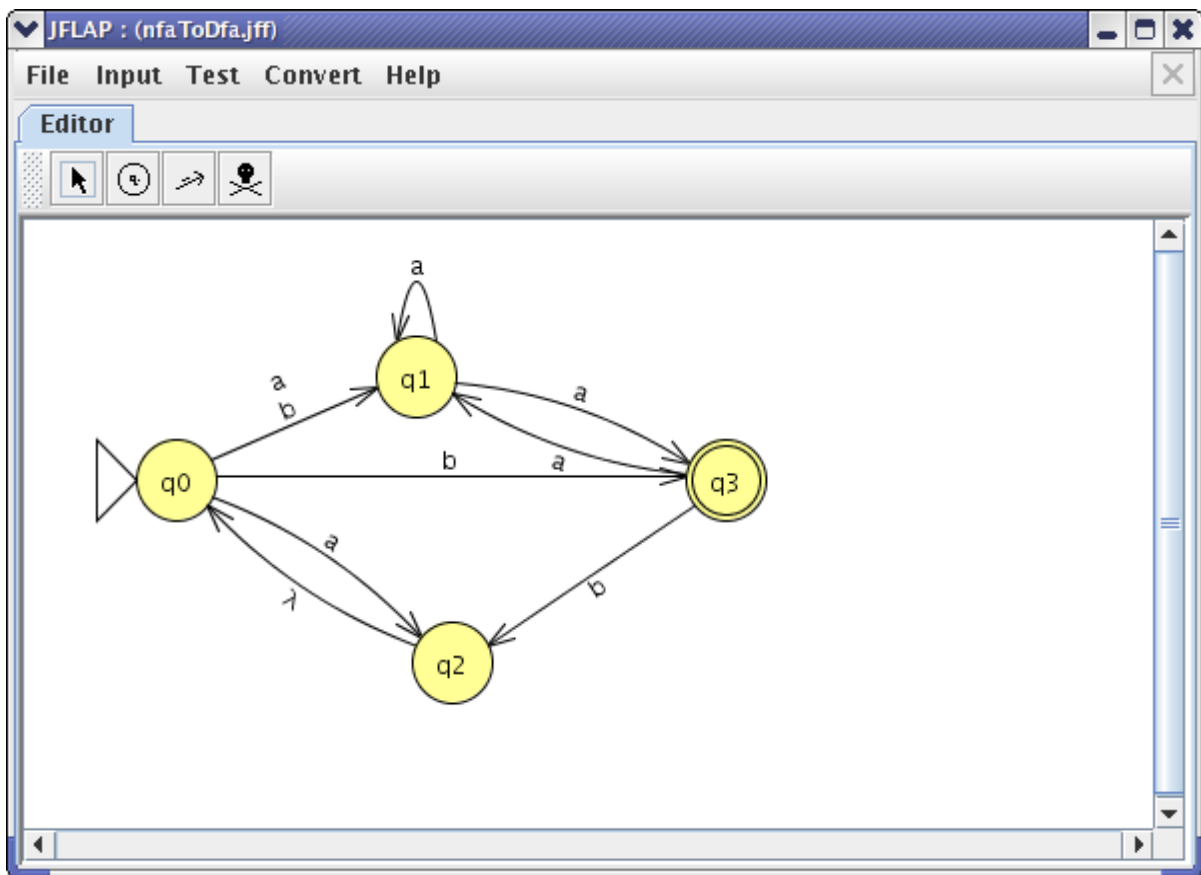
## EXPERIMENT 12

### Converting a NFA to a DFA

#### Introduction

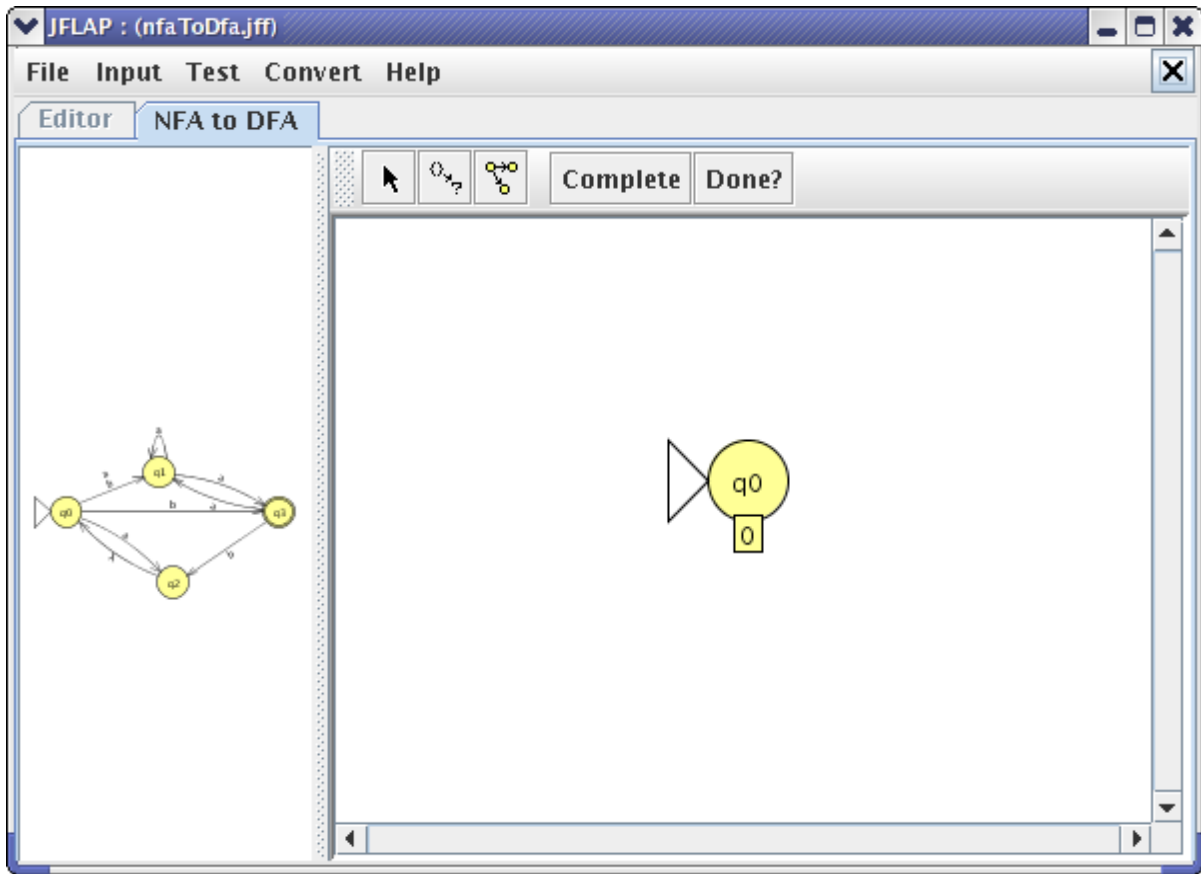
It is recommended, if you haven't already, to read the tutorial about [creating a finite automaton](#), which covers the basics of constructing a FA. This section specifically describes how one may transform any nondeterministic finite automaton (NFA) into a deterministic automaton (DFA) by using the tools under the “Convert → Convert to DFA” menu option.

To get started, open JFLAP. Then, either load the file [nfaToDfa.jff](#), or construct the nondeterministic finite automaton present in the screen below. When finished, your screen should look something like this:

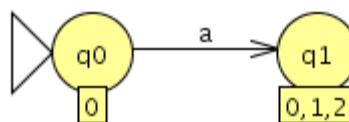


## Converting to a DFA

We will now convert this NFA into a DFA. Click on the “Convert → Convert to DFA” menu option, and this screen should come up:



The NFA is present in the panel on the left, and our new DFA is present in the screen to the right. Let's begin building our DFA. First, hold the mouse over the second button in the toolbar from the left to see the description of the button, which should be “Expand Group on (T)erminal”. Click on that button, and then click and hold down the mouse on state  $q_0$ . Then, drag the mouse away from the state, and release it somewhere in the white part of the right panel. Enter “a” when you are prompted for the terminal, and “0,1,2” when prompted for the group of NFA states. When finished, your screen should contain something like this (Note: one may adjust the states so that the layout resembles this image):

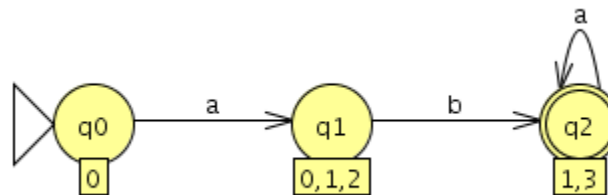


You are probably curious about what exactly we just did. Basically, we just built the first transition in

our DFA. In our NFA, we start at state “q0”. This is represented in the DFA through the “0” label in DFA state “q0”. The DFA's “a” expansion represents what happens when processing an “a” in the NFA. However, in the NFA there are three possible states we can proceed to when processing an “a”, NFA states “q0” and “q1”, and “q2”. This multiplicity of possibilities is represented by the label under DFA state “q1”, which is “0,1,2” (standing for NFA states “q0”, “q1”, and “q2”).

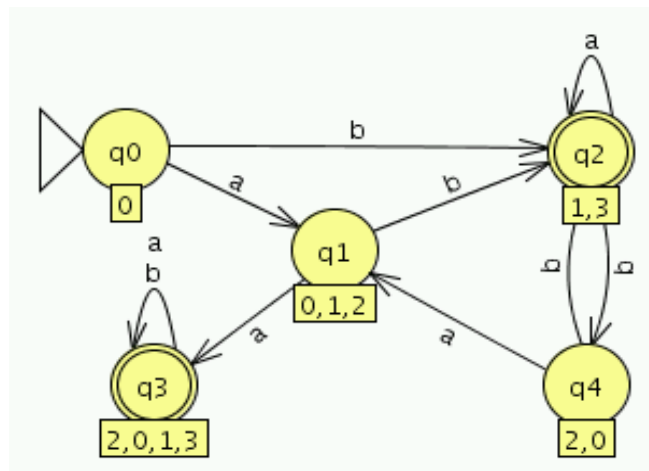
You should note that it is not possible to expand a state on a terminal that does not have a corresponding path in the NFA. For example, if one tries the same method above on state “q0”, but instead uses the terminal “c”, an error message will pop up.

Now we can add two more transitions in the DFA. Through the same method as before, add another expansion, starting from “q1”. The terminal value will be “b” and the states will be “1,3”. This will create a new state “q2”. Now, add an expansion starting from “q2”. This time, however, release the mouse on “q2” and enter the terminal “a”. When finished, something resembling the following should be present on your screen:



From NFA state “q1”, there are no expansions leading from it that contain a “b”. From NFA states “q0” and “q2”, a “b” will allow the program to reach NFA states “q1” or “q3”. Thus, the label for DFA state “q2” is “1,3”. You also might notice that DFA state “q2” is a final state. Since NFA state “q3” is a final state, and DFA state “q2” represents a possible path to NFA state “q3”, DFA state “q2” is a final state. It does not matter that DFA state “q2” also represents a possible path to NFA state “q1”, a nonfinal state. As long as one possible path represented by a DFA state is final, the DFA state is final. In the case of the expansion from “q2” to “q2”, since in DFA state “q2” one can be at NFA states “q1” or “q3”, and since processing any “a” from there will result in being in one of those two NFA states, the expansion does not result in a DFA state change.

Let's check and see if we are done. Go ahead and click on the “Done?” button. The message will tell us that we still have some work to do, as we have two more states and seven more transitions unaccounted for in the DFA. Go ahead and finish it. One can utilize, if desired, two other buttons in the toolbar. The “(S)tate Expander” button, if pressed, will fill out all transitions and create any new DFA states those transitions require when you click on an existing DFA state. Alternatively, if one wants to see the whole DFA right away, one can click on the “Complete” button. If either of these are pressed, the layout manager may not put everything onto the part of the screen currently visible, so one may have to maximize the window or find all states using the sliders. After finishing the DFA, and perhaps after moving the states around a little, you should have a picture resembling the following:



Now click the “Done?” button again. After a message informing you that the DFA is fully built, a new editor window is generated with the DFA in it. Congratulations, you have converted your NFA into a DFA!

### Course Materials References:

1. Compilers: Principles, Techniques, and Tools. 2nd edition by Aho, Lam, Sethi, Ulman.
2. The Compiler Design Handbook: Optimizations and Machine Code Generation by Y.N. Srikant P.Shankar.
3. Engineering a compiler by Keith Cooper, Linda Torcozon, 2nd Edition.
4. Compiler Construction: Principles and Practice by Kenneth C. Louden.
5. Optimizing compilers for modern architectures by Allen and Kennedy.