Lexical Analysis- Part 3

Lexical Analyzer Generator (Lex/flex)

Recap – Lexical Analysis

- What is lexical analysis?
- Why should LA be separated from syntax analysis?
- Tokens, patterns, and lexemes
- Difficulties in lexical analysis
- **Specification of tokens** - regular expressions and regular definitions
- **Recognition of tokens** - finite automata and transition diagrams
- Variant of finite automata (transition diagrams to represent patterns)
- Implementing a lexical analyzer from transition diagrams
- **LEX** - A Lexical Analyzer Generator

Lexical Analyzer Generator (Lex/flex)

Combining transition diagrams to form LA

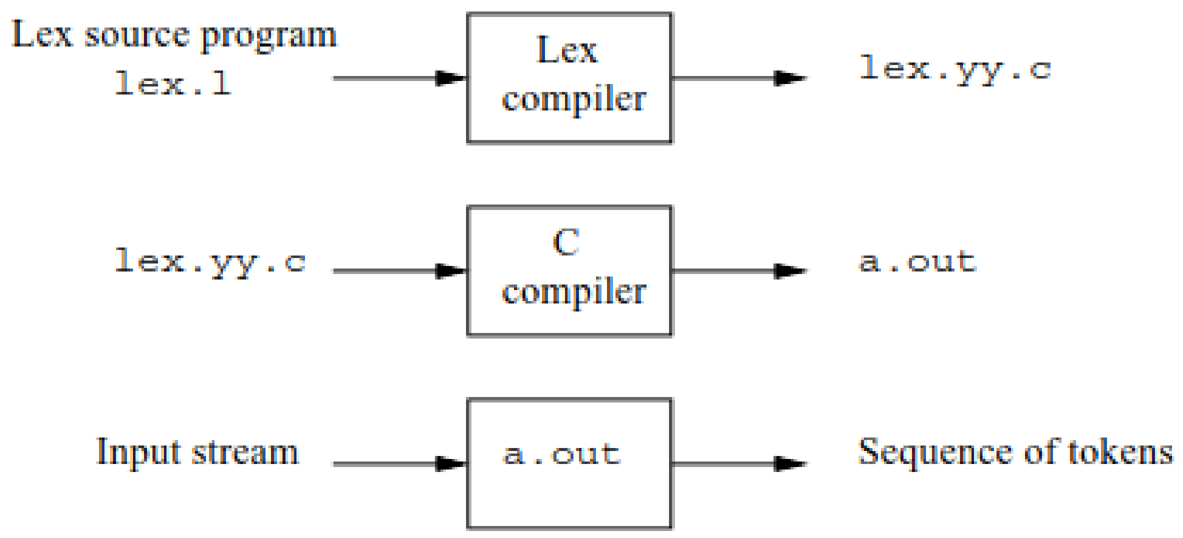
- Different transition diagrams must be combined appropriately to yield an Lexical Analyzer
  - Combining TDs is not trivial
  - It is possible to try different transition diagrams one after another
  - For example, TDs for reserved words, constants, identifiers, and operators could be tried in that order
  - However, this does not use the "**longest match**" characteristic (*thenext* would be an identifier, and not reserved word *then* followed by identifier *ext*)
  - To find the ***longest match***, all TDs must be tried and the longest match must be used

Using LEX to generate a lexical analyzer makes it easy for the compiler writer

LEX – A Lexical Analyzer Generator

- Lex has a language for describing regular expressions
- ***Lex language***: Allows specifying regular expressions to ***describe patterns*** for tokens.
- It generates a **lexical analyzer** (a ***pattern matcher*** for the regular expression specifications provided to it as input)
- Transforms patterns into transition diagrams and generates code.

How to use Lex- Creating a lexical analyzer using Lex



- Input source program in *Lex language* e.g., “lex.l”
- Commands to create an LA
  - **lex** lex.l – creates a C-program lex.yy.c

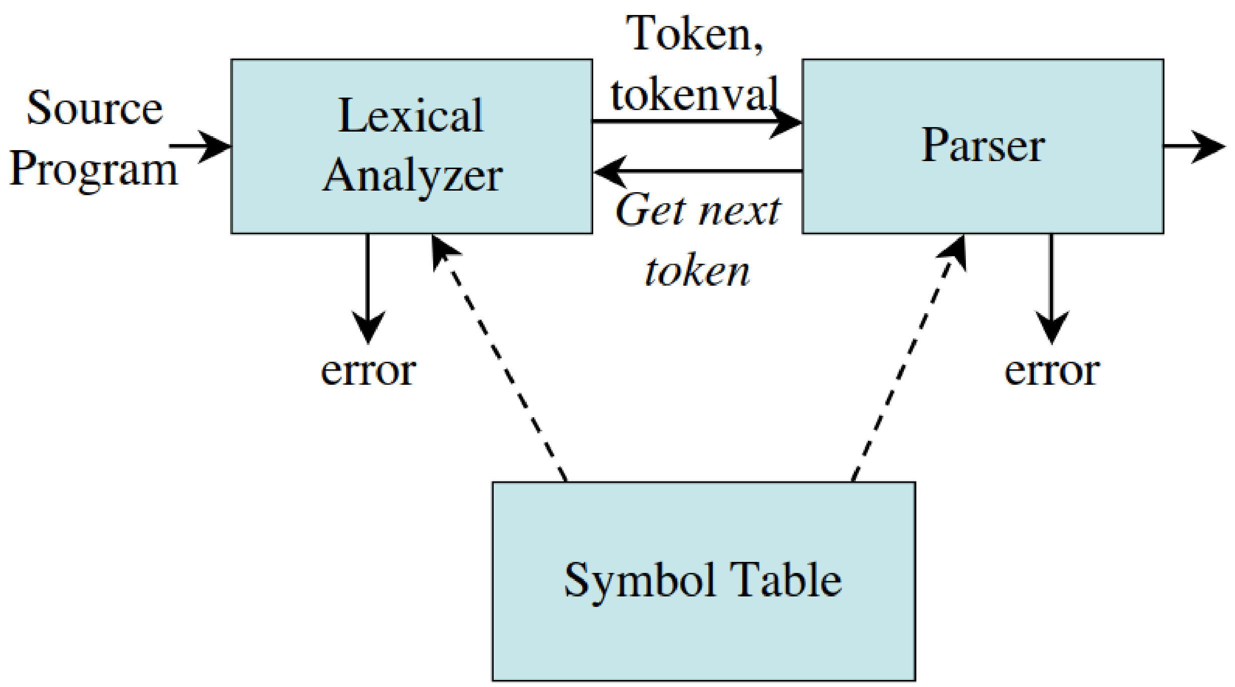
General structure of a Lex program

1. **declarations**
   - Declaration of ***vars***, ***constants*** (e.g., declared to represent name of a token),
   - **regular definitions**

2. **rules**
   - Each transition rule have the form: Pattern { Action }
   - Each pattern is an RE (may use regular definitions in declarations)
   - **Action**: Fragment of code typically written in C

3. **auxiliary functions**
   - Holds additional functions used in actions

Interaction of the lexical analyzer with the parser



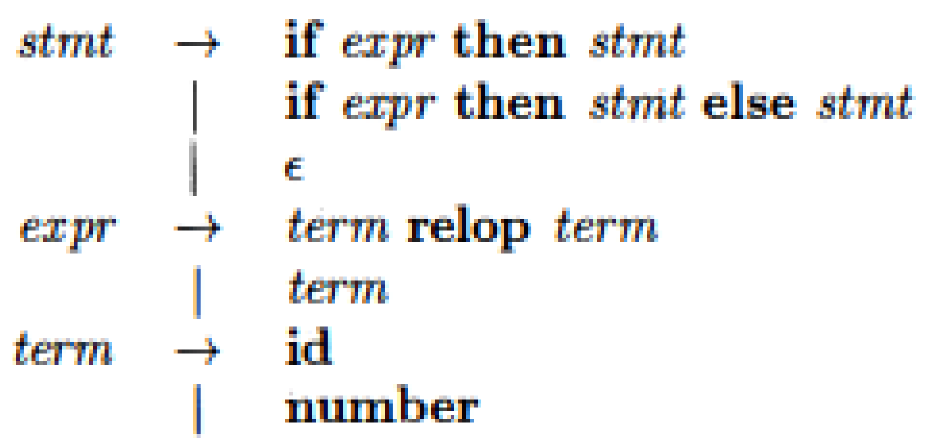
Interaction of the lexical analyzer with the parser

- The ***C*** program that is generated by **Lex** is compiled (which is the lexical analyzer).
- It is ***used as a subroutine*** by the parser
- It is a ***C* function that returns an integer** (representing code of one of the possible token names)
- **Shared global variables** between lexical analyzer and parser (*attribute values*, *pointer to symbol table*,..)

Interaction of the lexical analyzer with the parser

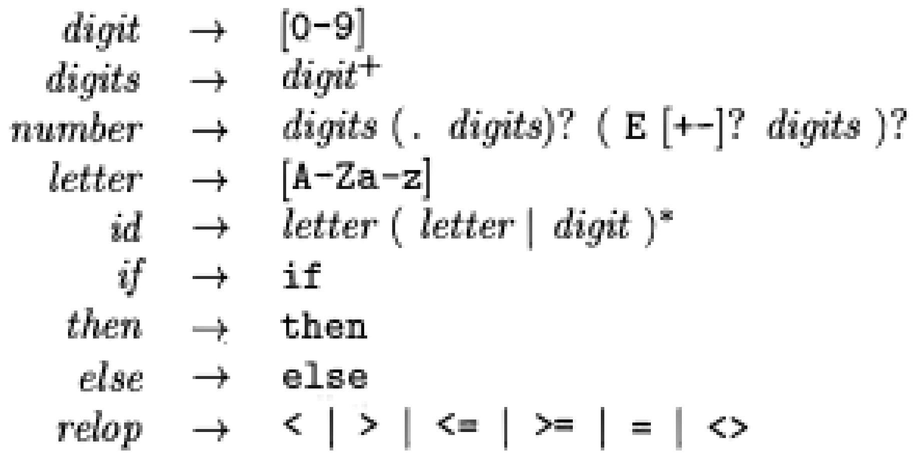
- Lexical analyzer when called by the parser
  - Begins reading **remaining input** character by character
  - Upon finding the longest prefix of the input that matches one of the patterns $\mathbf{P}_{i,}$
    - Executes the associated action $\mathbf{A}_i$
    - $\mathbf{A}_i$ typically returns to the parser
    - If it does not return (e.g., if $\mathbf{P}_i$ describes whitespace, comments..)
      - then it **continues to find additional lexemes** until one of the corresponding action causes a return to the parser.
    - Shared variable to pass additional information to the parser

Example: A Grammar for Branching Statements



$$\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term} \textbf{ relop } \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$

Grammar fragment describing a simple form of branching statements and conditional expressions



$$\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{[0-9]} \\
\mathit{digits} & \rightarrow & \mathit{digit}^{+} \\
\mathit{number} & \rightarrow & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \\
\mathit{letter} & \rightarrow & \texttt{[A-Za-z]} \\
\mathit{id} & \rightarrow & \mathit{letter}\ (\ \mathit{letter} \mid \mathit{digit}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{relop} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}$$

Patterns for tokens

Flex

- Takes a program written in a combination of Flex and C, and it writes out a file (called **lex.yy.c**) that holds a definition of function **yylex()**
  - `int yylex(void);`
- **yylex** reads from file stored in variable **yyin**
- **yytext** variable to store lexeme that is found
- **yyleng** length of lexeme

Example- FLEX program

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lex/Flex: Generating Lexical Analysers

Tool for generating scanners: programs which recognised lexical patterns in text

- **Lex input consists of 3 sections**:
  - regular expressions;
  - pairs of regular expressions and C code;
  - auxiliary C code.
- When the lex input is compiled, it generates as output a C source file **lex.yy.c** that contains a routine **yylex().**
- After compiling the C file, the executable will start isolating tokens from the input according to the regular expressions, and, for each token, will execute the code associated with it.

flex Example

```
%{
#define ERROR -1
int line_number=1;
%}
whitespace    [ \t]
letter        [a-zA-Z]
digit         [0-9]
integer       ({digit}+)
l_or_d        ({letter}|{digit})
identifier    ({letter}{l_or_d}*)
operator      [-+*/]
separator     [;,(){}]
%%
{integer}     {return 1;}
{identifier}  {return 2;}
{operator}|{separator}  {return (int)yytext[0];}
{whitespace}  {}
\n            {line_number++;}
.             {return ERROR;}
%%
int yywrap(void) {return 1;}
int main() {
    int token;
    yyin=fopen("myfile","r");
    while ((token=yylex())!=0)
       printf("%d %s \n", token, yytext);
    printf("lines %d \n",line_number);
}
```

Input file (“myfile”)

```
123+435+34=aaaa
329*45/a-34*(45+23)**3
bye-bye
```

flex Example

```
%{
#define ERROR -1
int line_number=1;
%}
whitespace    [ \t]
letter        [a-zA-Z]
digit         [0-9]
integer       ({digit}+)
l_or_d        ({letter}|{digit})
identifier    ({letter}{l_or_d}*)
operator      [-+*/]
separator     [;,(){}]
%%
{integer}     {return 1;}
{identifier}  {return 2;}
{operator}|{separator}  {return (int)yytext[0];}
{whitespace}  {}
\n            {line_number++;}
.             {return ERROR;}
%%
int yywrap(void) {return 1;}
int main() {
    int token;
    yyin=fopen("myfile","r");
    while ((token=yylex())!=0)
       printf("%d %s \n", token, yytext);
    printf("lines %d \n",line_number);
}
```

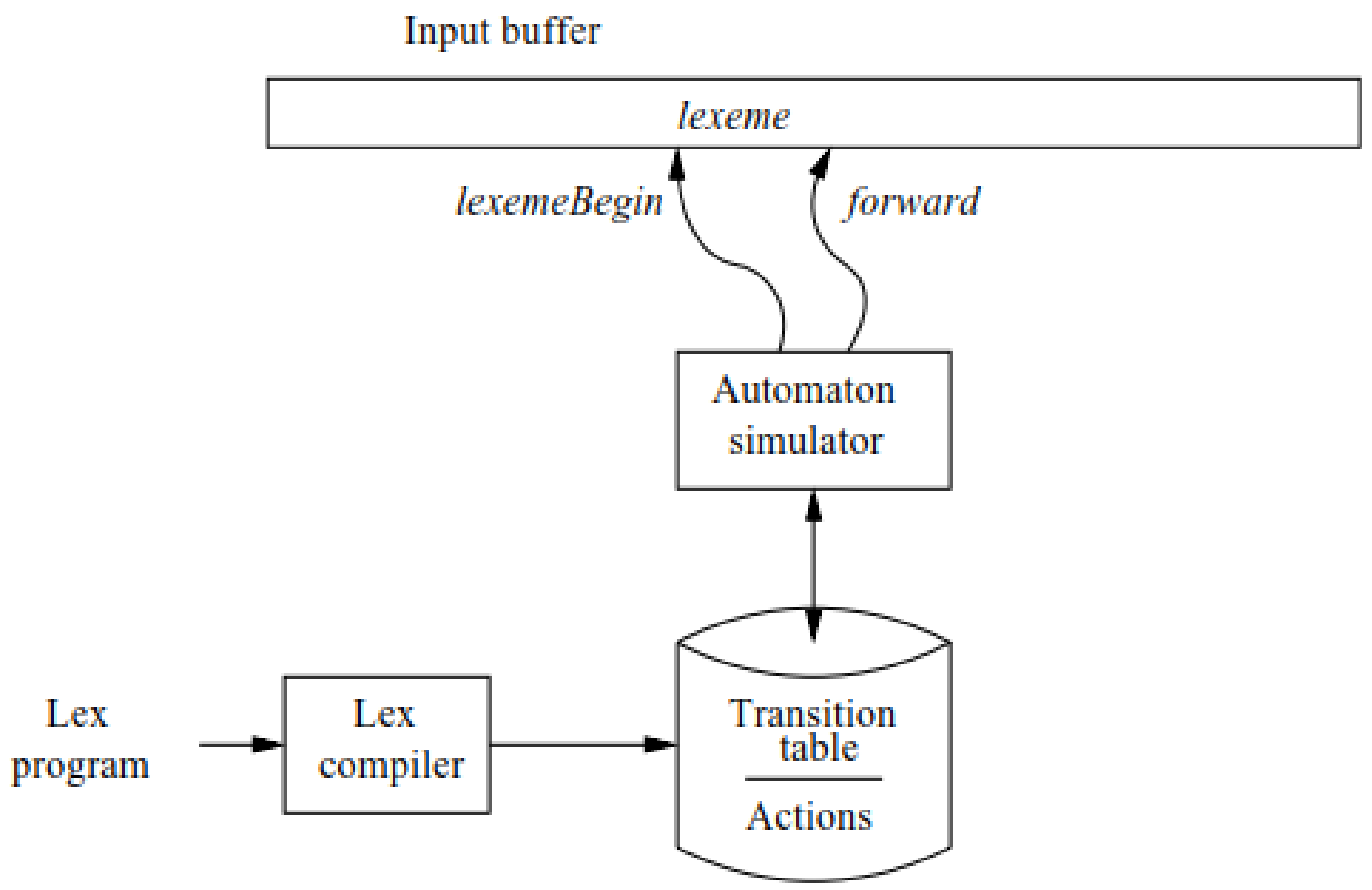
Input file (“myfile”)

```
123+435+34=aaaa
329*45/a-34*(45+23)**3
bye-bye
```

Output:

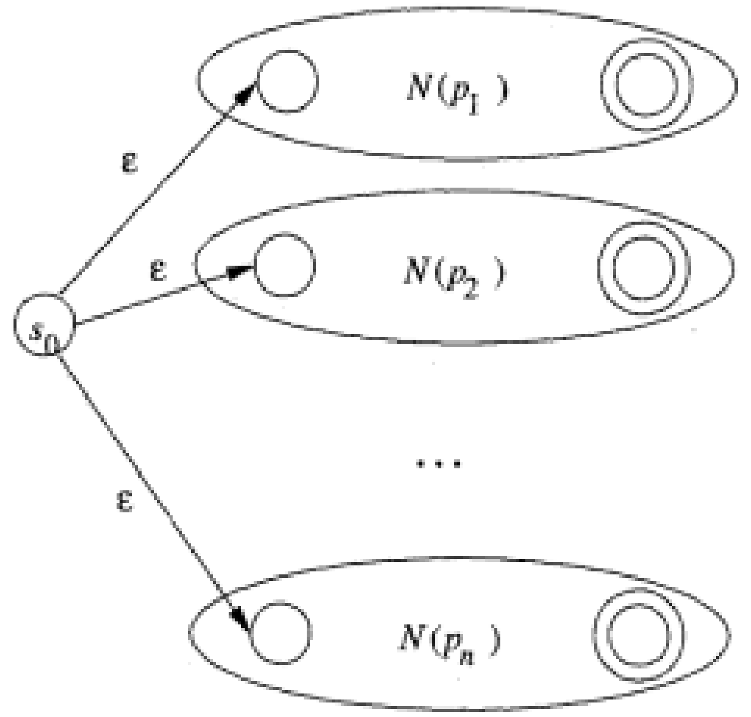
```
  1 123
 43 +
  1 435
 43 +
  1 34
 -1 =
  2 aaaa
  1 329
 42 *
  1 45
 47 /
  2 a
 45 -
  1 34
 42 *
 40 (
  1 45
 43 +
  1 23
 41 )
 42 *
 42 *
  1 3
  2 bye
 45 -
  2 bye
lines 4
```

Structure of the generated lexical analyzer



Example

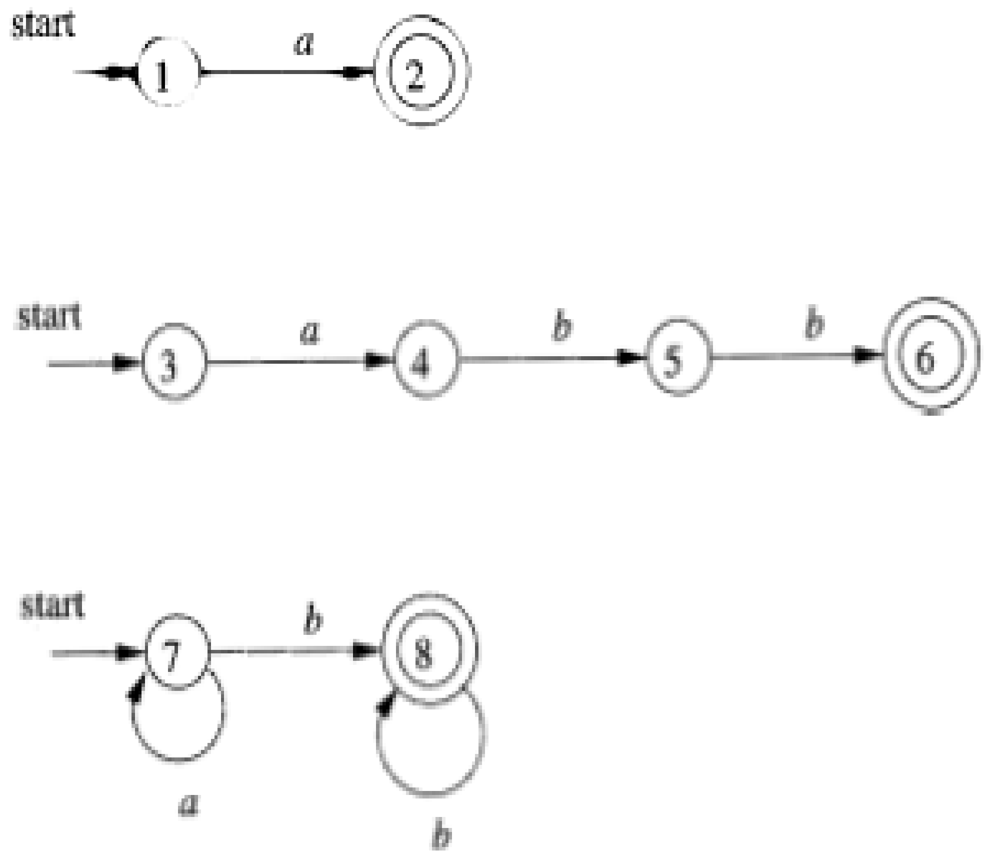
$\mathbf{a}$	{ action A_1 for pattern p_1 }
$\mathbf{abb}$	{ action A_2 for pattern p_2 }
$\mathbf{a^*b^+}$	{ action A_3 for pattern p_3 }



NFA constructed by Lex

NFAs

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
$\mathbf{a}^*\mathbf{b}^+$	{ action A_3 for pattern p_3 }



Simulation, Pattern matching

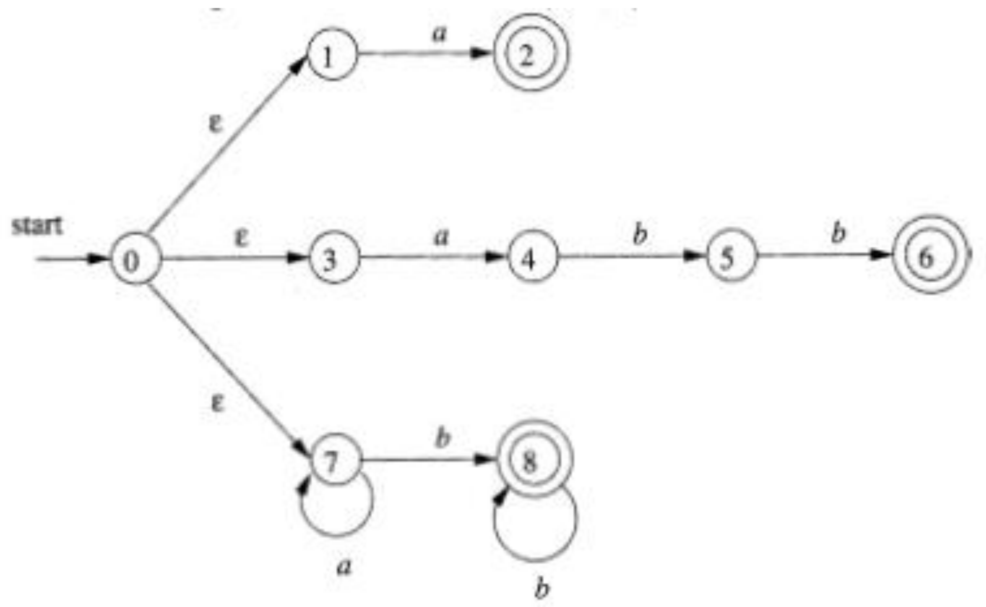


Figure 3.52: Combined NFA

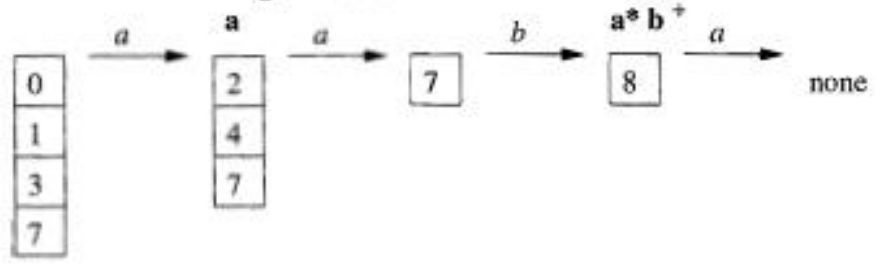
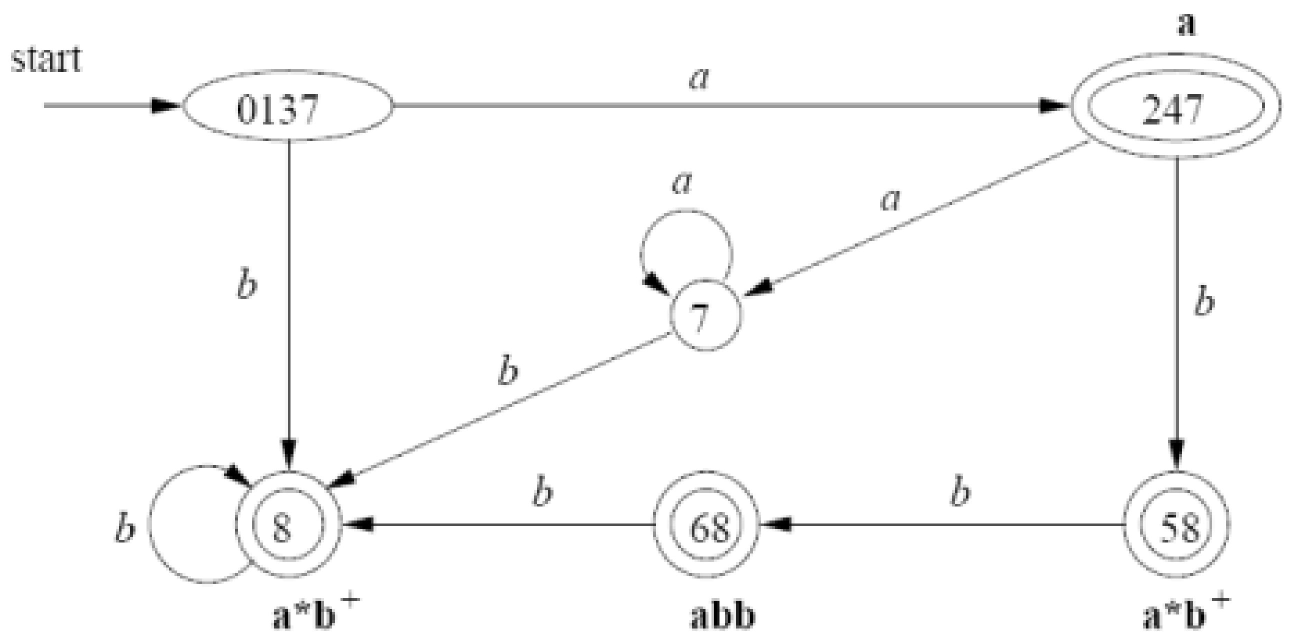


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

Consider Input:
aaba

LA Based on DFA



start
0137
247
a
7
8
68
58
b
a*b+
abb
a*b+