# CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

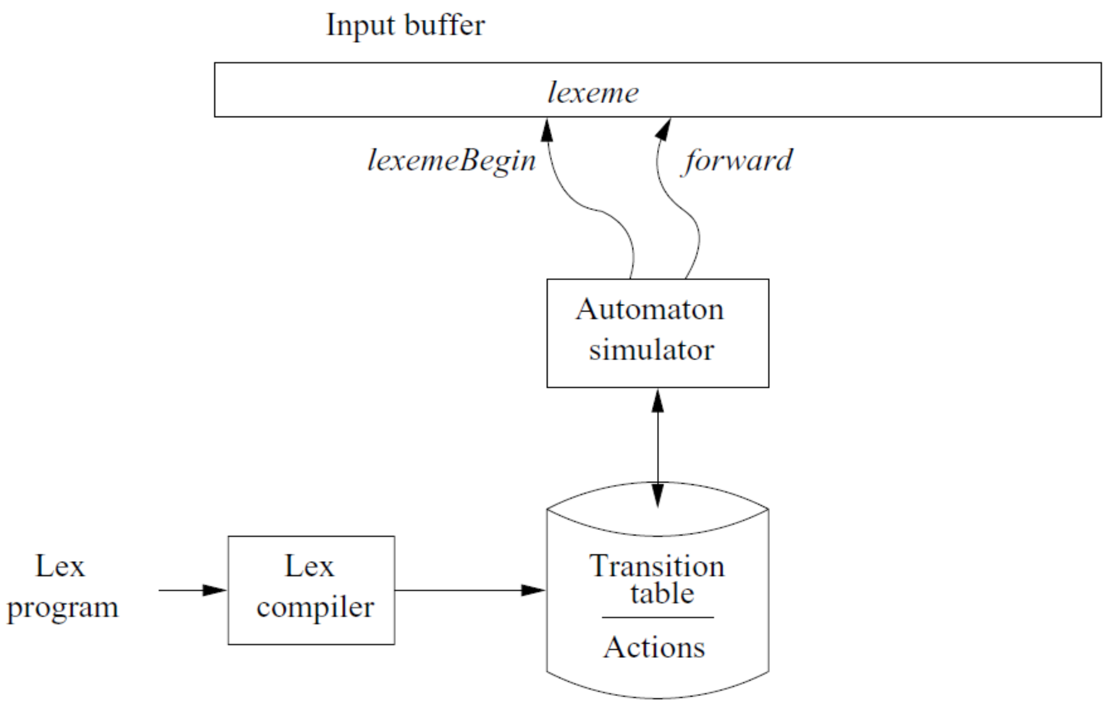
- **Design of a Lexical-Analyzer Generator (3.8)**
  - **3.8.1: The Structure of the Generated Analyzer**
  - **3.8.2: Pattern Matching Based on NFA’s**
  - **3.8.3: DFA’s for Lexical Analysis**
- **Syntax Analysis (Ch4) (We will begin from the new chapter in the next time)**

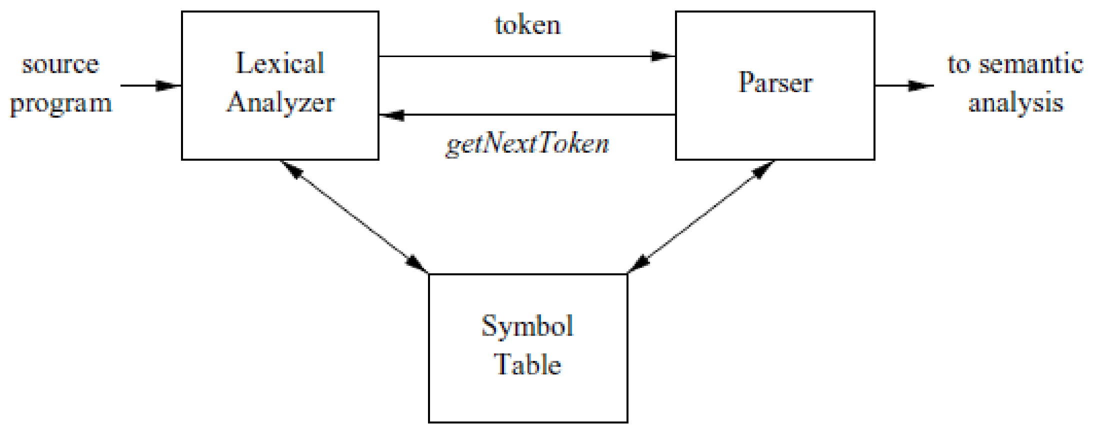
## 3.8.1 The structure of the generated analyzer

- Previously, in Section 3.7, we had seen the idea how a lexical-analyzer generator such as *Lex,* is architected.
- We had discussed 2 approaches, based on NFA and DFA about the architecting of lexical-analyzer generator
- The latter (DFA) is essentially the implementation of *Lex*
- As we had introduced earlier, Lex is not just a tool but also has its own language / structure

## 3.8.1 The Structure of the Generated Analyzer

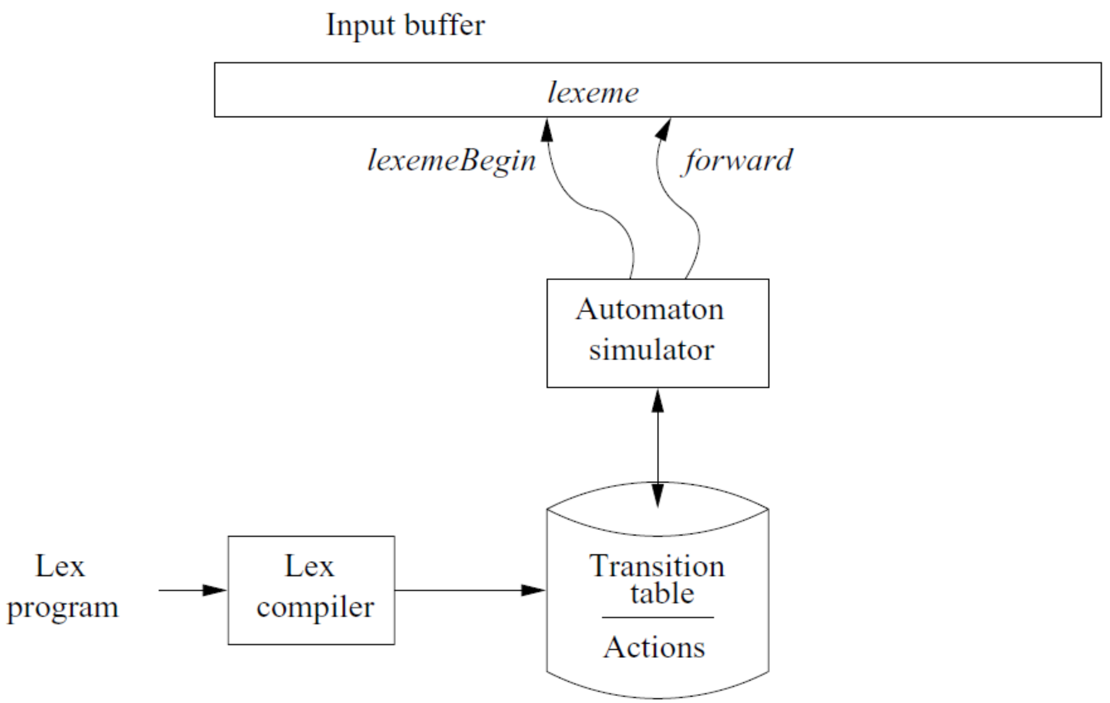
- Here is the overview of the **architecture** of a lexical analyzer generated by *Lex*.
- The program that serves as the lexical analyzer **includes a fixed program** that **simulates** an **automaton** --- **Automation Simulator**

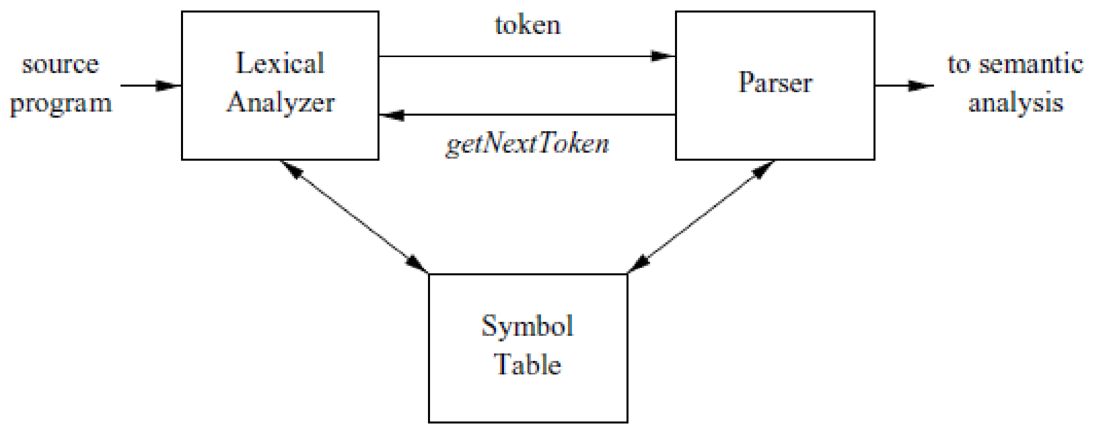




## 3.8.1 The Structure of the Generated Analyzer

- So basically, the figure of LHS is roughly equivalent to Lexical Analyzer (LA) box on RHS,
- But why it is called Automation Simulator?





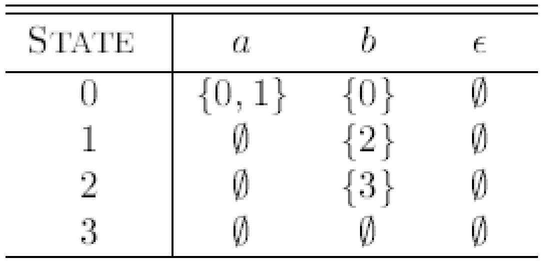
## 3.8.1 The Structure of the Generated Analyzer

**The purpose of this is to minimize the human inputs and is originated from using the software to control machinery. Now is used in AI/ML or manufacturing area**

- **Answer**:
- The lexical analyzer can help us in identifying the **error** by using the, 1) **automation machine** and, 2) the **grammar** of the given language (i.e. C, C++) and gives **row** number and **column** number of the error.
- Still remember the purpose of LA?
  - Tokenizing. Divide the code into valid tokens
  - Remove white space characters
  - Remove comments
  - Generating the error message by row/column number positioning (in your IDE)

# 3.8.1 The Structure of the Generated Analyzer

- Transition table is the thing we introduced earlier. (in Finite Automata)
  - It is a **tool** used to guide the transition diagram
  - It is used to represent the **transition**, based on the given info. on {states, input_symbol} pairs
- For the interactions, on the upper part of the figure, we had already covered previously
  - We run the automation (transition diagram), and based on this, depending on the cases, we can **control and move the two pointers**



| STATE | $a$ | $b$ | $\epsilon$ |
|---|---|---|---|
| 0 | $\{0,1\}$ | $\{0\}$ | $\emptyset$ |
| 1 | $\emptyset$ | $\{2\}$ | $\emptyset$ |
| 2 | $\emptyset$ | $\{3\}$ | $\emptyset$ |
| 3 | $\emptyset$ | $\emptyset$ | $\emptyset$ |

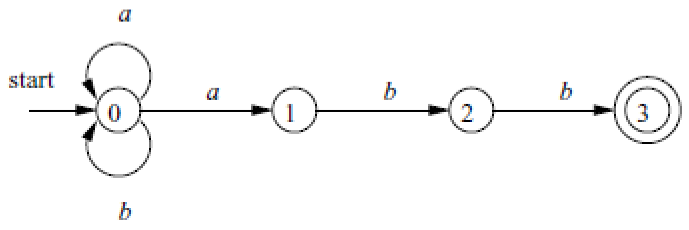
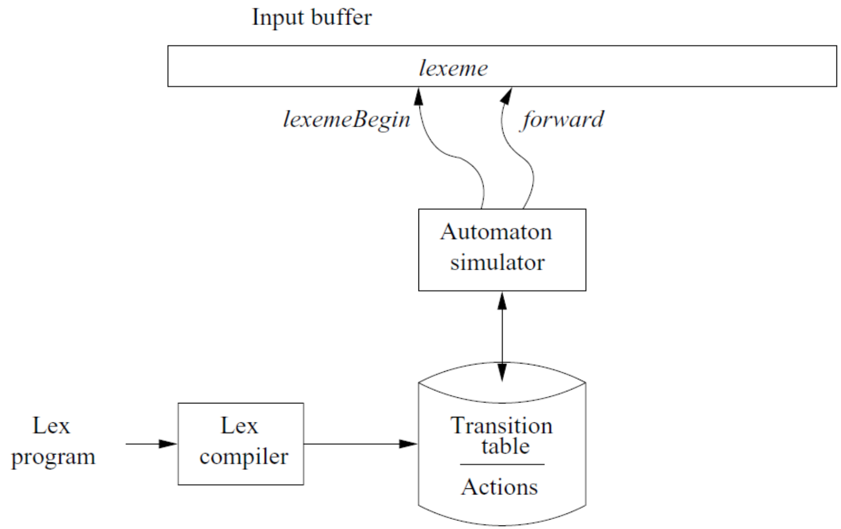


Figure 3.24: A nondeterministic finite automaton



## 3.8.1 The Structure of the Generated Analyzer

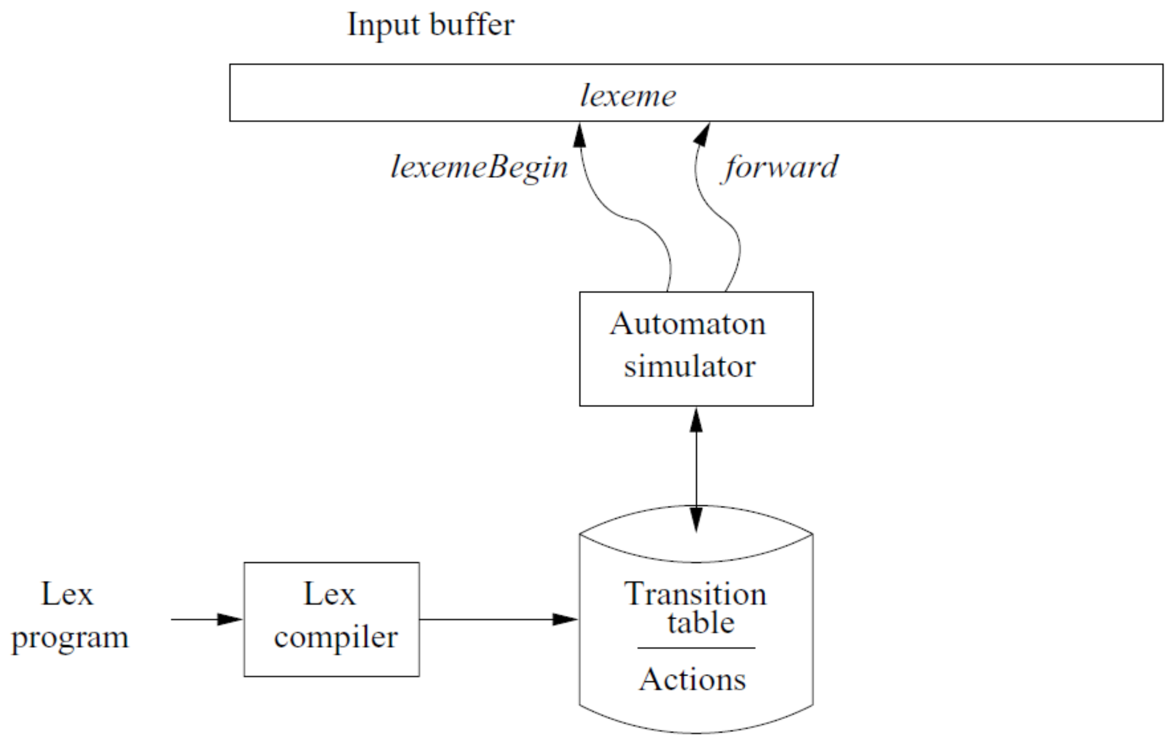
- To construct the automation, we begin by taking each regular-expression pattern in the *Lex* program and converting it to NFA.
- Still remember the *Lex?*

A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

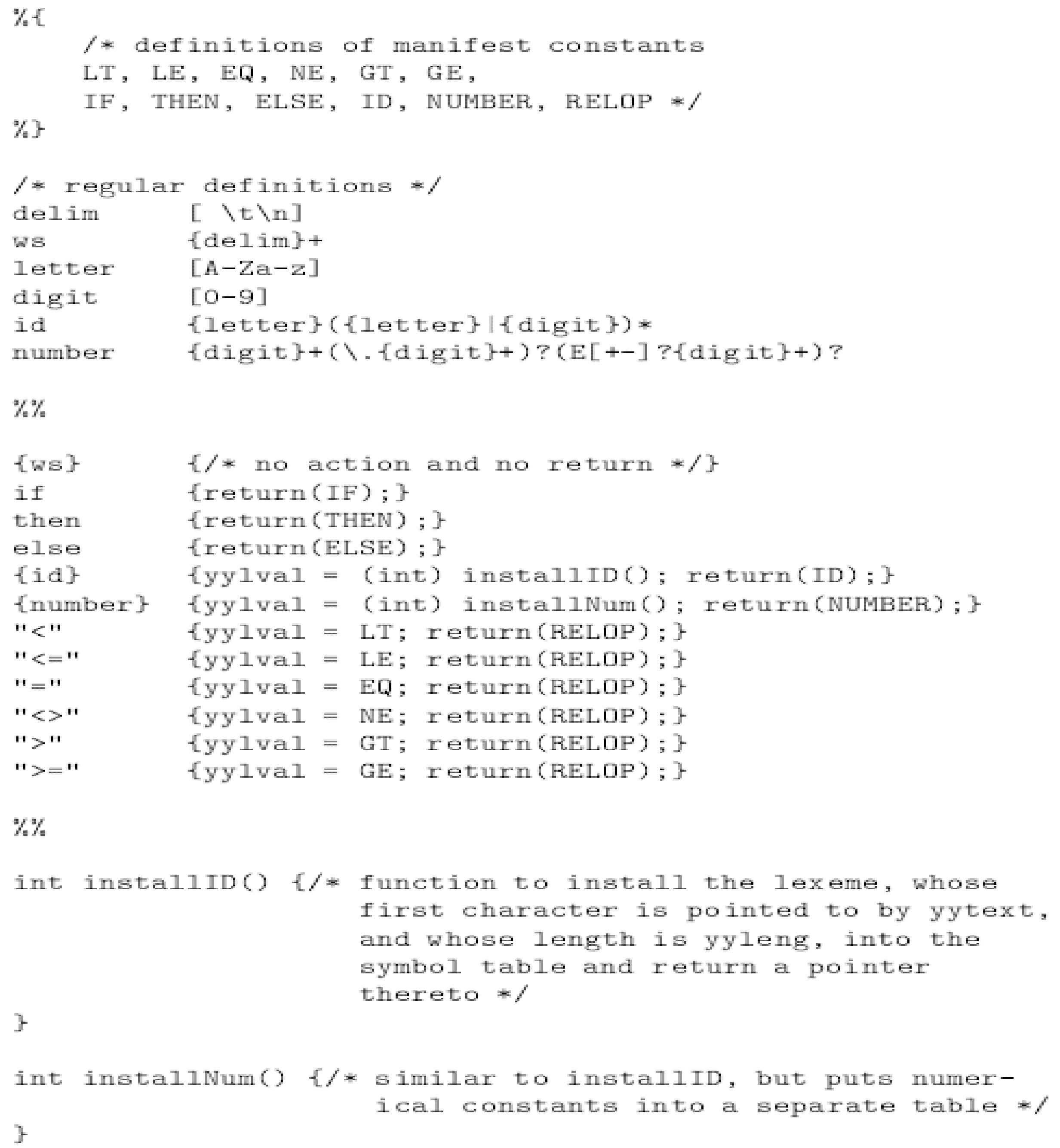
The translation rules each have the form

```
Pattern   { Action }
```



## 3.8.1 The Structure of the Generated Analyzer

- **An example from the book.**
- ***Lex is*** **used to tokenize by using the translation**
- **Now let's go back to the construction of automation**



```
%{
    /* 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 */
}
```

Figure 3.23: Lex program for the tokens of Fig. 3.12

## 3.8.1 The Structure of the Generated Analyzer

- In order to do the construction of automation, we need a single automaton that will recognize lexemes matching any of the patterns in the program
- Because each regular-expression pattern in the *Lex* program will be converted into NFA, by using the idea introduced in Ch 3.7, we combine all the NFA's into one by introducing **a new start state** with “**epsilon-transitions**” to each of the (original) start states of the (different) NFA's Ni for pattern pi.
- It might be confusing for the first time to look at it. See the next page for detail

## 3.8.1 The Structure of the Generated Analvzer

- 3 patterns (regular expressions)
  - p1, p2 and p3 (which means, pi)
- Note that the *Lex* rule is to take the **longest matching**, so we continue in reading b's, until another “a” is met. (i.e. abb)
- Matching has the “**order**” (of pattern)
  - i.e. “abb” is matched in 2nd or 3rd pattern. We still need to consider it a lexeme in p2

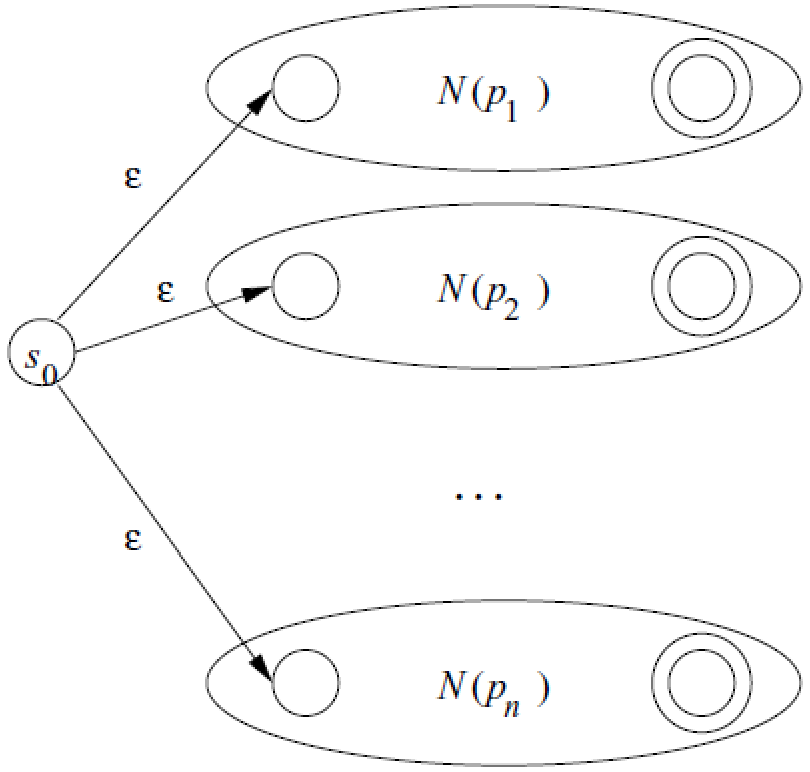


Figure 3.50: An NFA constructed from a Lex program

| | |
|---|---|
| **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 $p_3$ } |

# 3.8.1 The Structure of the Generated Analyzer

- This is the 3 NFA's that recognize the 3 patterns. (Fig. 3.5.1)
- Fig. 3.52 shows these 3 NFA‘s combined into a single NFA by the addition of **start 0** and **three epsilon-transitions**

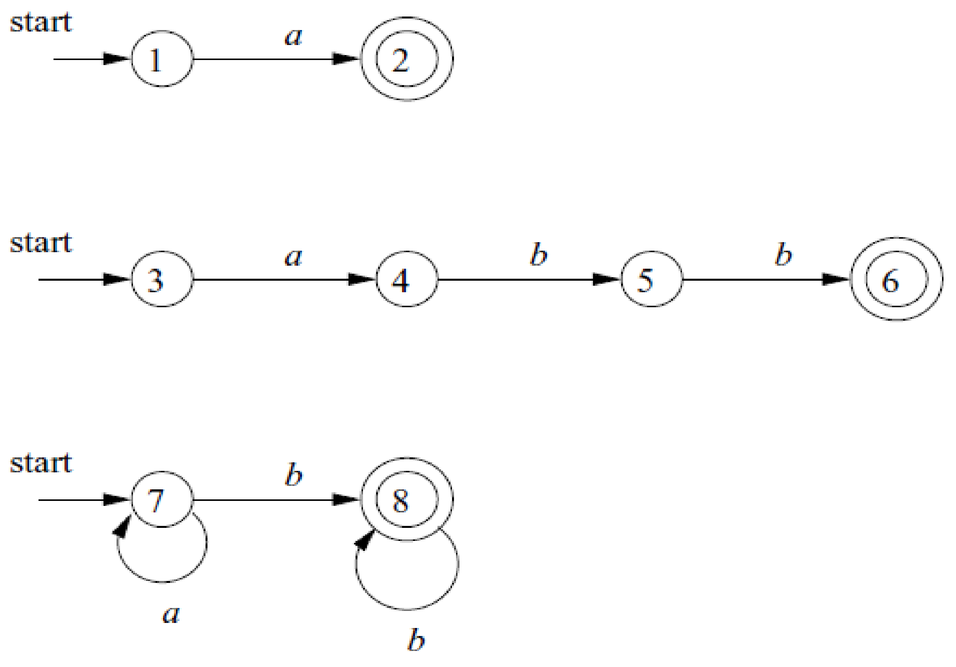


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

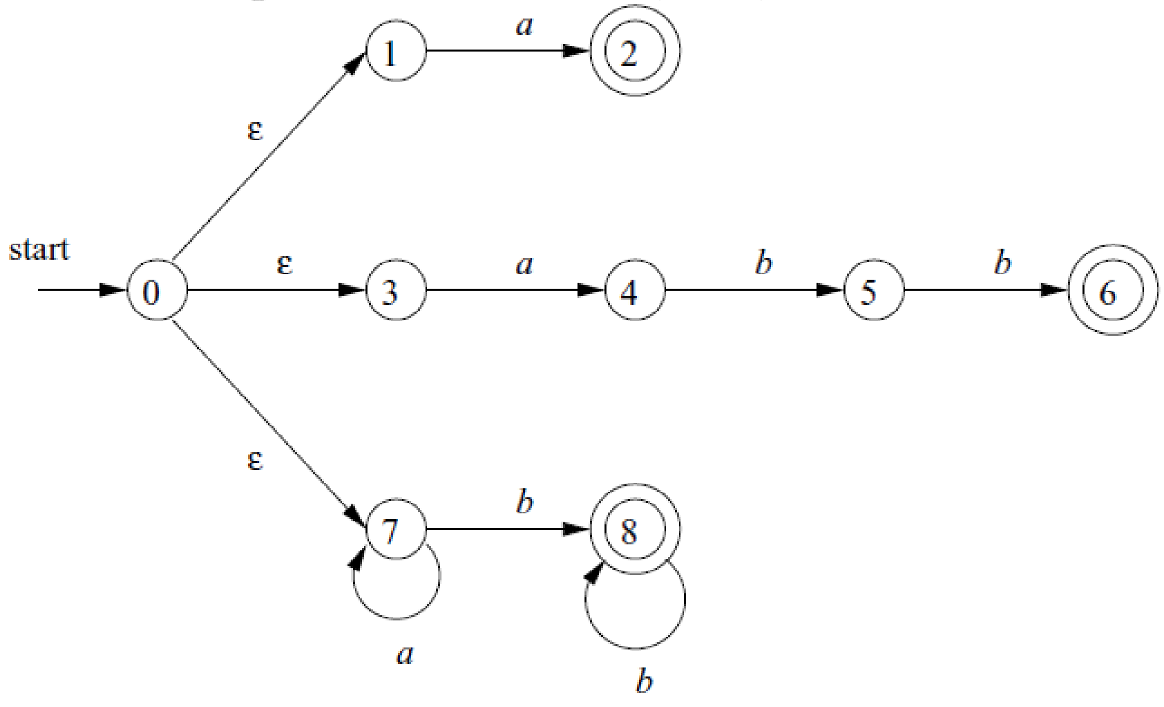


Figure 3.52: Combined NFA

## 3.8.2 Pattern Matching Based on NFA's

- If the **lexical analyzer** simulates an NFA such as that of Fig. 3.52, then it must read input beginning at the point on its input which we have referred to as lexemeBegin
- As it moves the pointer (called forward ahead) in the input, it calculates the set of states it is in, at each point.

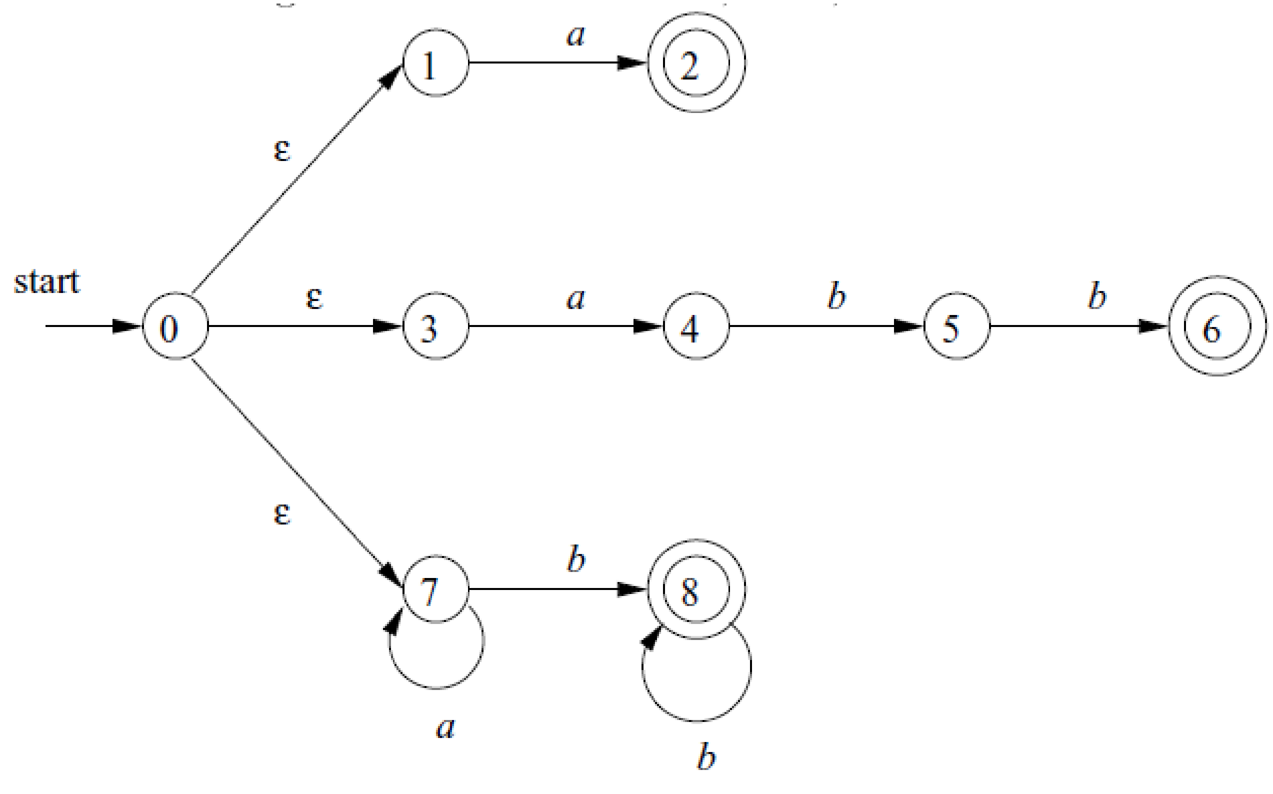


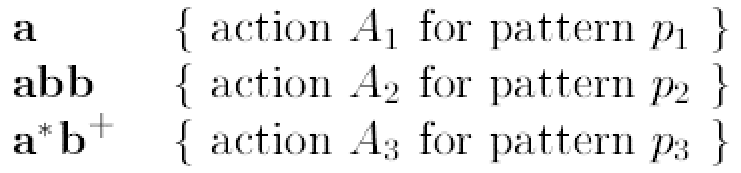
Figure 3.52: Combined NFA

## 3.8.2 Pattern Matching Based on NFA's

- Eventually, the NFA simulation reaches a point on the input where there are no more next states
- That means, there won't be any longer prefix of the input would ever get the NFA to an accepting state
- So now, we are ready to decide on the "longest prefix", that is a **lexeme** matching **some pattern**
- Why we need the lexeme? We need the token! Lexeme is a part of token remember?

# 3.8.2 Pattern Matching Based on NFA's

- Here is the detail of an algorithm for pattern matching, an example, provided by the book, talking about how the input string "aaba" is processed
- It will eventually conclude that "aab" is the longest prefix that gets us to an accepting state.
- We therefore select "aab" as the lexeme and execute actin A3 which should include a return to the parser indicating that **the token whose pattern is p3**=a*b+, has been found
- Let's take a look into detail! (See the next page)



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

## 3.8.2 Pattern matching Based on NFA's

- It begins with the start state 0, the algorithm can be run concurrently in different paths.
- The input begins with "aaba"
- Ride the epsilon closure transitions, the 1st state set is {0, 1, 3, 7}
- We now ride the transition of 1st "a", aaba
- So it can bring us to {2, 4, 7}. However, 2 is the accepting state, so the pattern *a*, this 1st pattern has been matched!

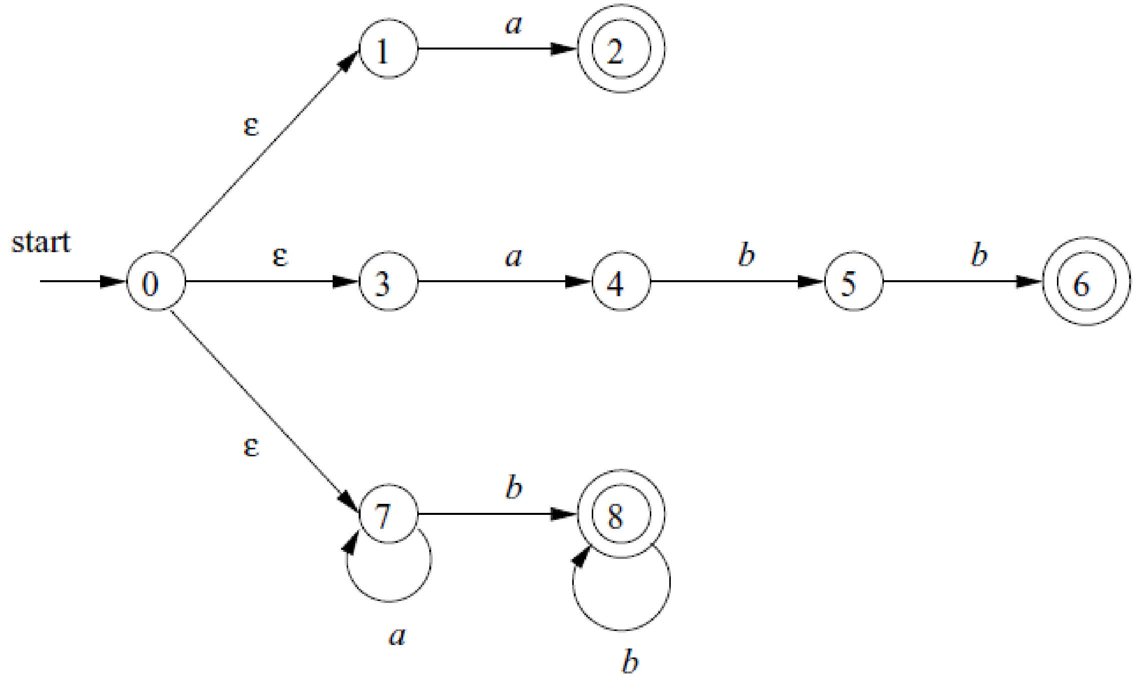


Figure 3.52: Combined NFA

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

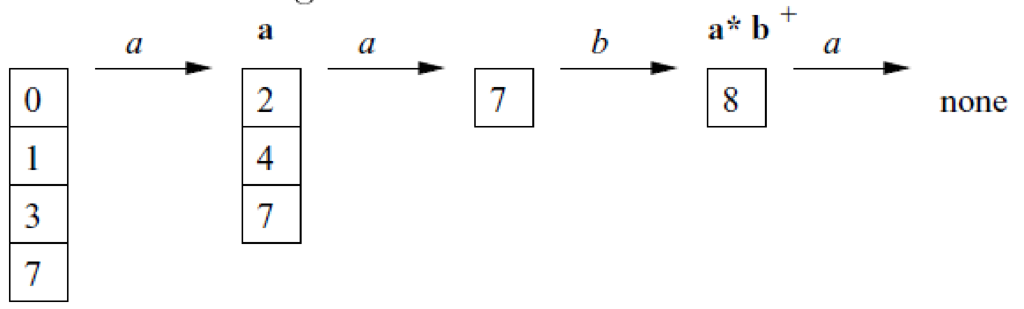


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

## 3.8.2 Pattern matching Based on NFA’s

- We now read something further, the transition of 2nd “a”, aaba
- We previous are standing on the states, 2, 4, 7, after reading one more “a”? No transitions “a” available for the path on the top. Similarly, no transition for the path in the middle. What about the bottom one? It can keep doing the self-looping!
- So, after taking the ride of 2nd “a”, it will bring us to state “7” ! For the previous 2 paths, the top and middle one are the “dead end”
- After the reading of aaba, we can only move the bottom one, from state 7 to state 8.
- (Cont. onto the next page)

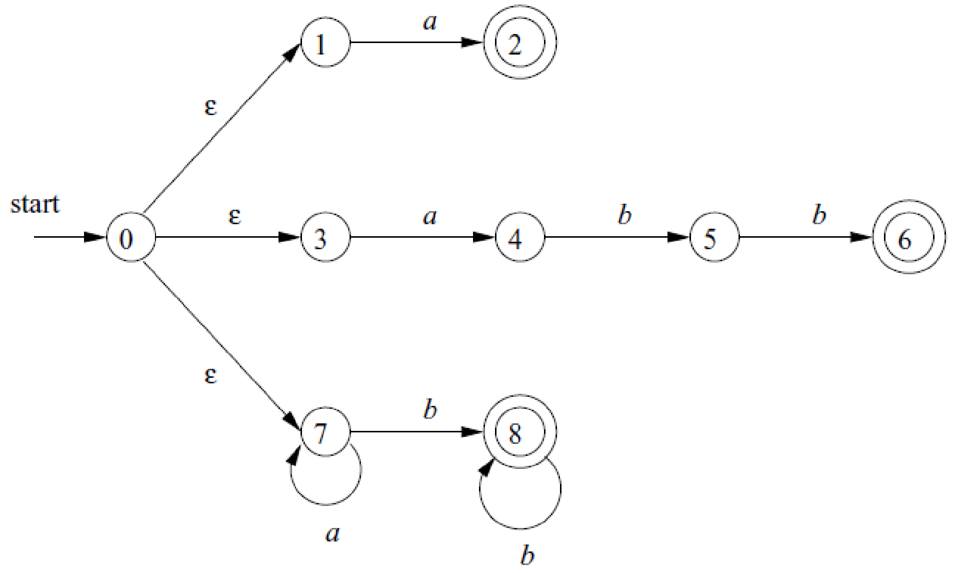
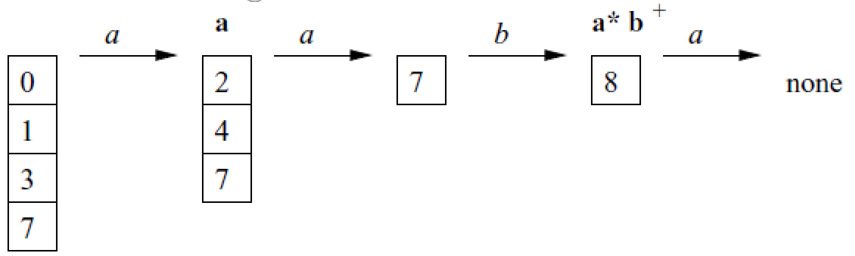


Figure 3.52: Combined NFA



| | |
|---|---|
| **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 $p_3$ } |

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

# 3.8.2 Pattern matching Based on NFA's

- Fortunately, state 8 is the accepting state, which indicates the 3rd pattern a*b+ is matched
- Finally we process the last "a", aaba. Because "8" is the last one we have, which is already accepting, after a transition of "a" is read in, we have no way to go! So there will be a none in Figure 3.53
- Finally, "**aab**" is selected as the **lexeme** and **A3** is executed

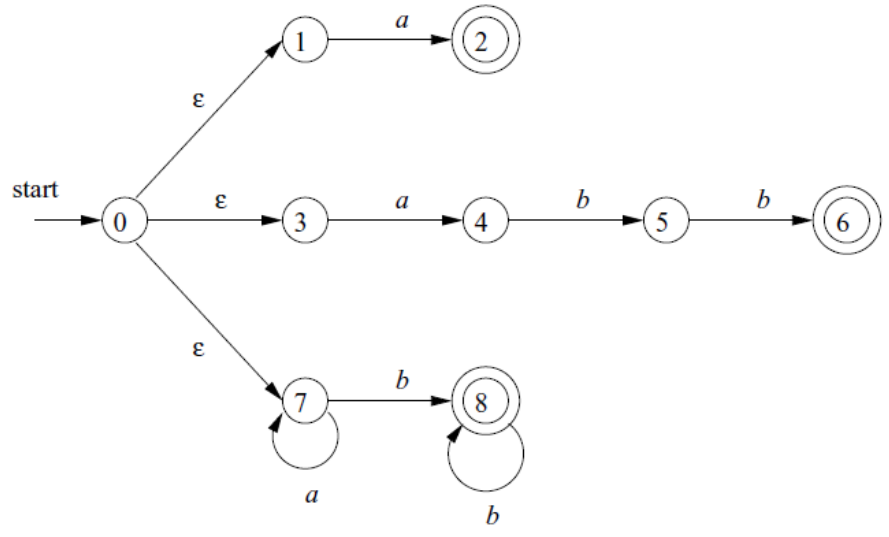


Figure 3.52: Combined NFA

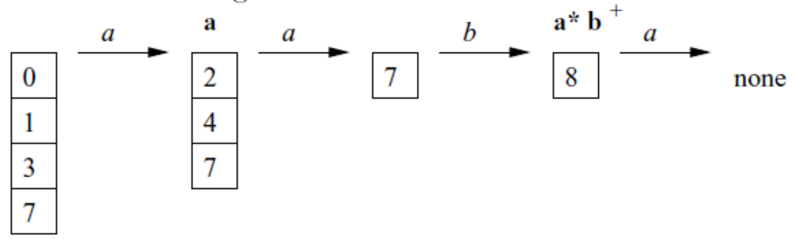


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

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

## 3.8.3 DFA’s for Lexical Analyzers

- Let’s play the game by using transition graph for one more time!
- There’s another architecture (DFA for LA), resembling the output of Lex, is to convert NFA for all the patterns into an equivalent DFA, using the Algorithm 3.20 we covered in the previous class
- The point is, within each of the DFA state, if there are one or more accepting NFA states, we need to determine the **first pattern** whose accepting state is represented, and make that **pattern** as the **output** of the DFA **state**.
  - Hard to understand isn’t it? I will give you the example in next page

## 3.8.3 DFA's for Lexical Analyzers

- This transition diagram is based on DFA and is constructed from NFA in Fig. 3.52, by using the subset construction algorithm we covered previously
- The accepting states are labeled by the pattern that is identified by that state
- For example, state "68" has 2 accepting states, mapping to the pattern "abb" and "a*b+"
- Since the "abb" is listed first, "abb" is the pattern associated with the state 68

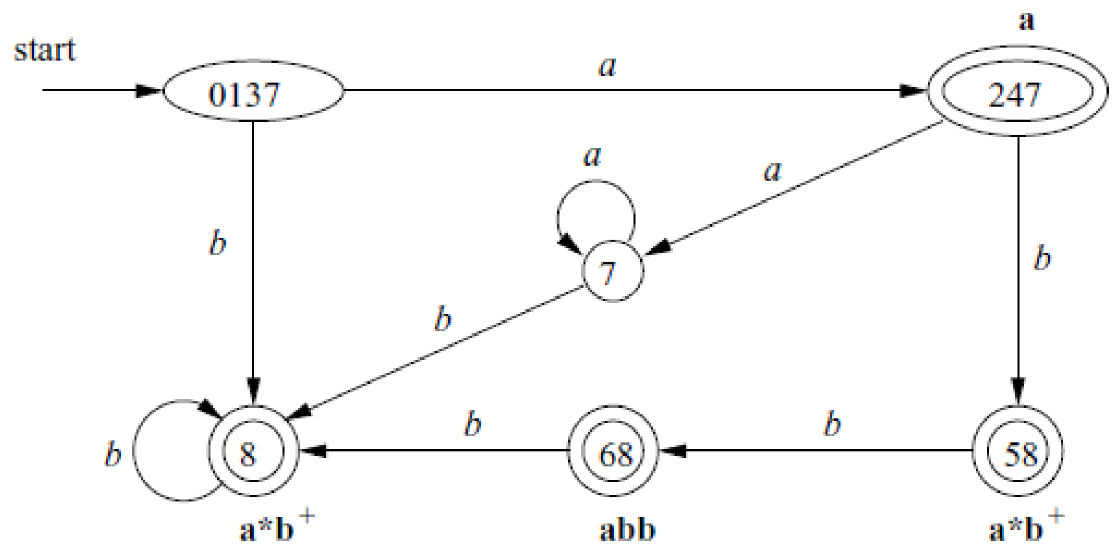


Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

# 3.8.3 DFA's for Lexical Analyzers

- Here is an example in the book
- Given the input "abba" and how do we identify the patterns based on this transition diagram?
- If we have a "walk", from "0137", the states we entered is 0137, 247, 58, 68 (Then, done!? We only walked abba)
- The final "a" has no transition out of state 68. Now we consider 68 is an accepting state. So, we report the pattern p2=abb

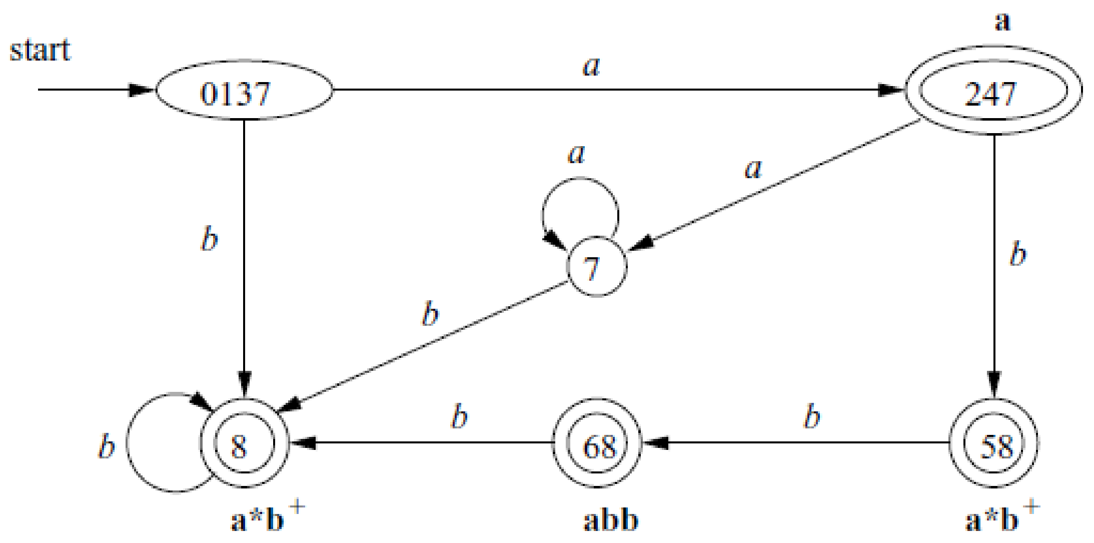


Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and $\mathbf{a^*b^+}$