Unit - I | Part-2

OVERVIEW OF COMPILATION

➢ The Lexical-Analyzer Generator: Lex

- Use of Lex
- Structure of Lex Programs
- Conflict Resolution in Lex
- The Lookahead Operator

➢ Design of a Lexical-Analyzer Generator

- The Structure of the Generated Analyzer
- Pattern Matching Based on NFA's
- DFA's for Lexical Analyzers
- Implementing the Lookahead Operator

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

The Lexical-Analyzer Generator Lex

- **`Lex/flex`**: A Language for Specifying Lexical Analyzers
- Implemented by *Lesk* and *Schmidt* of *Bell Lab* initially for *Unix*
- Not only a table generator, but also allows "*actions*" to associate with Regular Expression's.
- **`Lex`** is widely used in the *Unix* community
- **`Lex`** is not efficient enough for production compilers, however.

The Lexical-Analyzer Generator Lex

- The input notation for the *Lex tool* is referred to as the *Lex language* and the tool itself is the *Lex compiler*.
- The *Lex compiler* transforms the input patterns into a transition diagram and generates code, in a file called **`lex.yy.c`**, that simulates this transition diagram.

Use of Lex

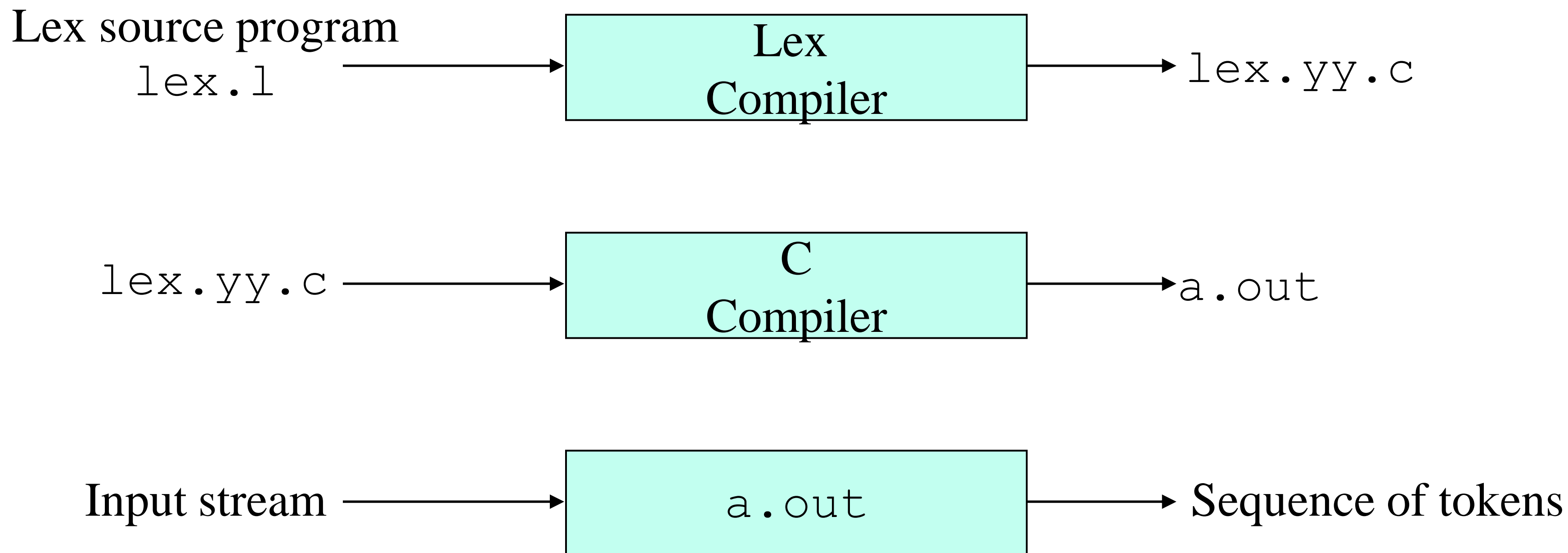

Figure: Creating a lexical analyzer with Lex

Use of Lex

- An input file, which we call **`lex.l`**, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms **`lex.l`** to a C program, in a file that is always named **`lex.yy.c`**.
- The latter file is compiled by the **`C`** compiler into a file called **`a.out`**, as always.
- The **`C-compiler`** output is a working *lexical analyzer* that can take a stream of input characters and produce a stream of tokens.

Use of Lex

- The normal use of the compiled **C** program, referred to as **a.out** in previous Fig. , is as a subroutine of the parser.
- It is a **C** function that returns an integer, which is a code for one of the possible token names.
- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable **yylval***, which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token.

*The **yy** that appears in **yylval** and **lex.yy.c** refers to the **Yacc** parser-generator, and which is commonly used in conjunction with **Lex**.

Structure of Lex Programs

- A **Lex** program has the following form:

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

Structure of Lex Programs

- The **declarations** section includes declaration of variables, *manifest constants* (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.
- The **translation rules** each have the form:

```
Pattern  { Action }
```

- Each pattern is a regular expression, which may use the regular definitions of the declaration section.
- The *actions* are *fragments of code*, typically written in **C**, although many variants of **Lex** using other languages have been created.

Structure of Lex Programs

- The third section **`auxiliary functions`** holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

Example: Lex program for the tokens

Lex program that recognizes the tokens shown below and returns the token found.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
< >	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Example: Lex program for the tokens

```
%{
/* 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}+)?
```

Example: Lex program for the tokens

```
%%
{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);}
%%
```

Example: Lex program for the tokens

```
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 numerical constants into a separate table*/
}
```

Example: Lex program for the tokens

The action taken when **`id`** is matched is threefold:

1. Function **`installID()`** is called to place the *lexeme* found in the *symbol table*.
2. This function returns a pointer to the *symbol table*, which is placed in global variable **`yylval`**, where it can be used by the *parser* or a later component of the compiler. *Note* that **`installID()`** has available to it two variables that are set automatically by the *lexical analyzer* that *Lex generates*:
   - **a)** **`yytext`** is a pointer to the beginning of the *lexeme*, analogous to *lexemeBegin*.
   - **b)** **`yyleng`** is the length of the *lexeme* found.
3. The token name **`ID`** is returned to the *parser*.

Conflict Resolution in Lex

We have suggest to the two rules that `Lex` uses to decide on the proper *lexeme* to select, when several prefixes of the input match one or more *patterns*:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more *patterns*, prefer the *pattern* listed first in the `Lex` program.

The Lookahead Operator

- **`Lex`** automatically reads one character ahead of the last character that forms the selected *lexeme*, and then retracts the input so only the *lexeme* itself is consumed from the input.
- However, sometimes, we want a certain *pattern* to be matched to the input only when it is followed by a certain other characters.
- If so, we may use the **`slash`** in a *pattern* to indicate the end of the part of the *pattern* that matches the *lexeme*.
- What follows **`/`** is additional *pattern* that must be matched before we can decide that the *token* in question was seen, but what matches this second *pattern* is not part of the *lexeme*.

The Lookahead Operator

`Example:`

In **`Fortran`** and some other languages, keywords are not reserved. That situation creates problems, such as a statement.

```
IF(I,J) = 3
```

where **`IF`** is the name of an array, not a keyword. This statement contrasts with statements of the form

`IF( condition ) THEN` ...

where **`IF`** is a keyword.

The Lookahead Operator

`Example:`

Fortunately, we can be sure that the keyword **`IF`** is always followed by a left parenthesis, some text-the condition-that may contain parentheses, a right parenthesis and a letter. Thus, we could write a **`Lex`** rule for the keyword **`IF`** like:

```
IF / \( .* \) {letter}
```

This rule says that the pattern the lexeme matches is just the two letters IF. The slash says that additional pattern follows but does not match the lexeme. In this pattern, the first character is the left parentheses. Since that character is a Lex metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning. The dot and star match “any string without a newline."

Design of a Lexical-Analyzer Generator

Outline:

- Design of a Lexical-Analyzer Generator
  - The Structure of the Generated Analyzer
  - Pattern Matching Based on NFA's
  - DFA's for Lexical Analyzers
  - Implementing the Lookahead Operator

Design of a Lexical-Analyzer Generator

- We shall apply the techniques presented previously (previous lecture) to see how a **`lexical analyzer generator`** such as **`Lex`** is architected.
- We discuss *two approaches*, based on **`NFA`**'s and **`DFA`**'s; the latter is essentially the implementation of **`Lex`**.

The Structure of the Generated Analyzer

- The following *figure* overviews the *architecture of a lexical analyzer* generated by **`Lex`**.

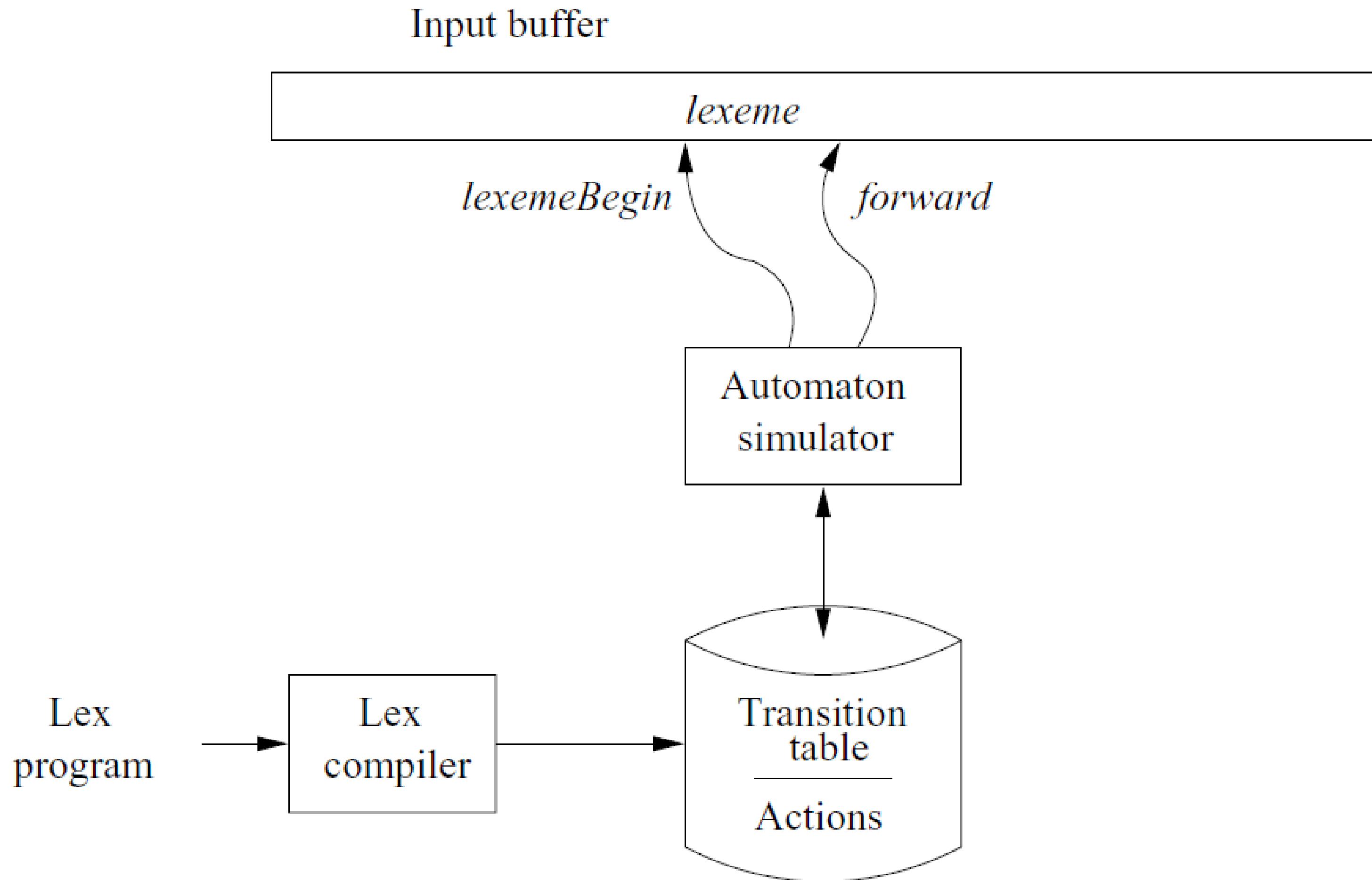

Fig: A Lex program is turned into a transition table and actions, which are used by a Finite-automaton simulator

The Structure of the Generated Analyzer

The components are

1. A **`transition table`** for the automaton.
2. Those functions that are passed directly through **`Lex`** to the output.
3. The **`actions`** from the input program, which appear as fragments of code to be invoked at the appropriate time by the **`automaton simulator`**.

The Structure of the Generated Analyzer

- To construct the automaton, we begin by taking each regular expression pattern in the **Lex** *program* and converting it to an **NFA**.
- We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the **NFA**'s into one by introducing a new start state with **ϵ-transitions** to each of the start states of the **NFA**'s N_i for pattern p_i. This construction is shown in below *Fig*.

The Structure of the Generated Analyzer

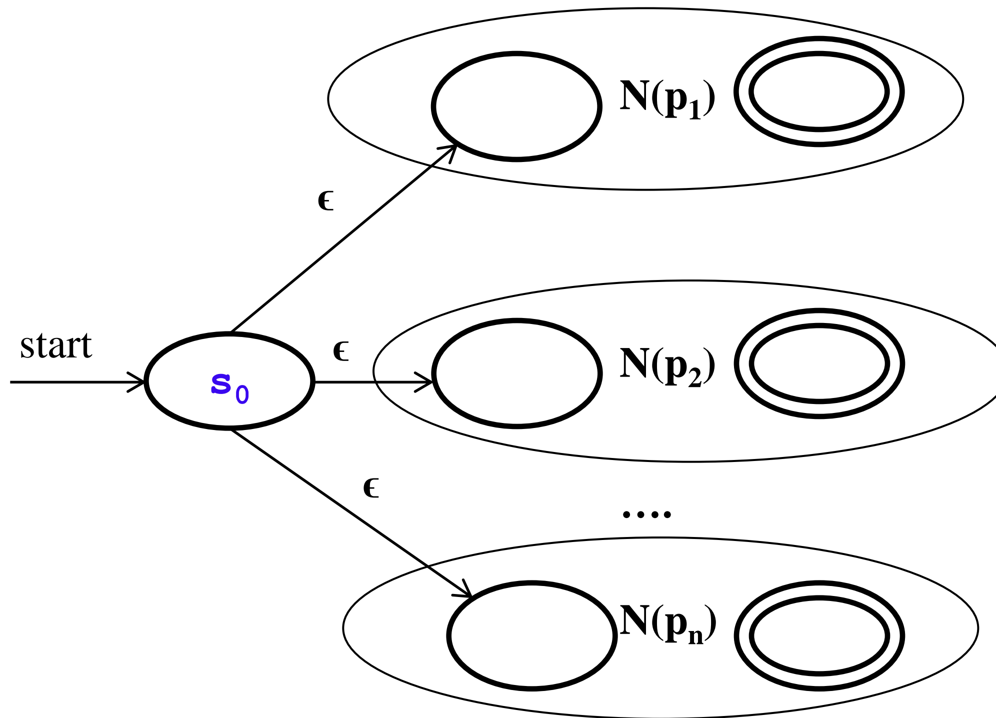

Fig: An **NFA** constructed from a **Lex program**

The Structure of the Generated Analyzer

We shall illustrate the ideas of this section with the following simple, *abstract example*:

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

The Structure of the Generated Analyzer

The following *Figure 1* shows three **NFA**'s that recognize the three patterns. Then, *Figure 2* shows these three **NFA**'s combined into a single **NFA** by the addition of *start state* **0** and *three* **ϵ-transitions**.

The Structure of the Generated Analyzer

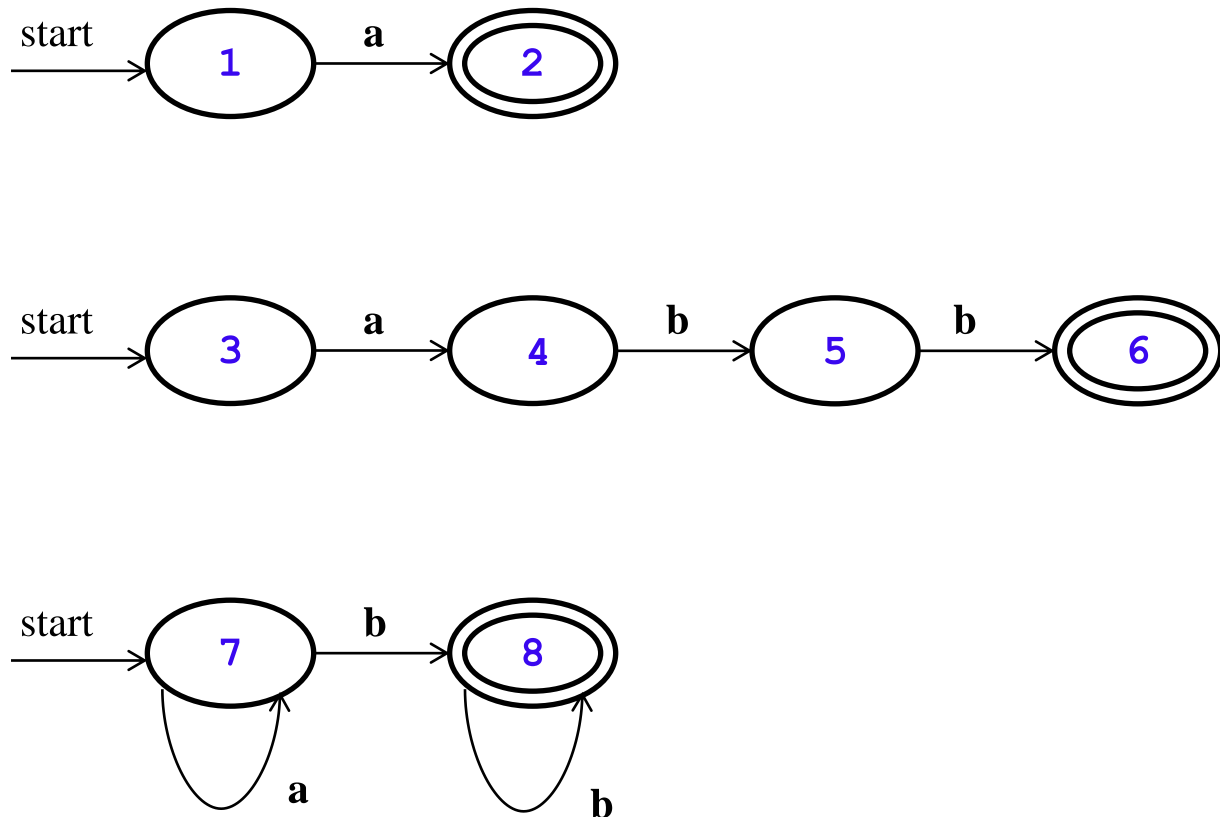

Figure 1: **NFA's** for **a**, **abb**, and $\mathbf{a^*b^+}$

The Structure of the Generated Analyzer

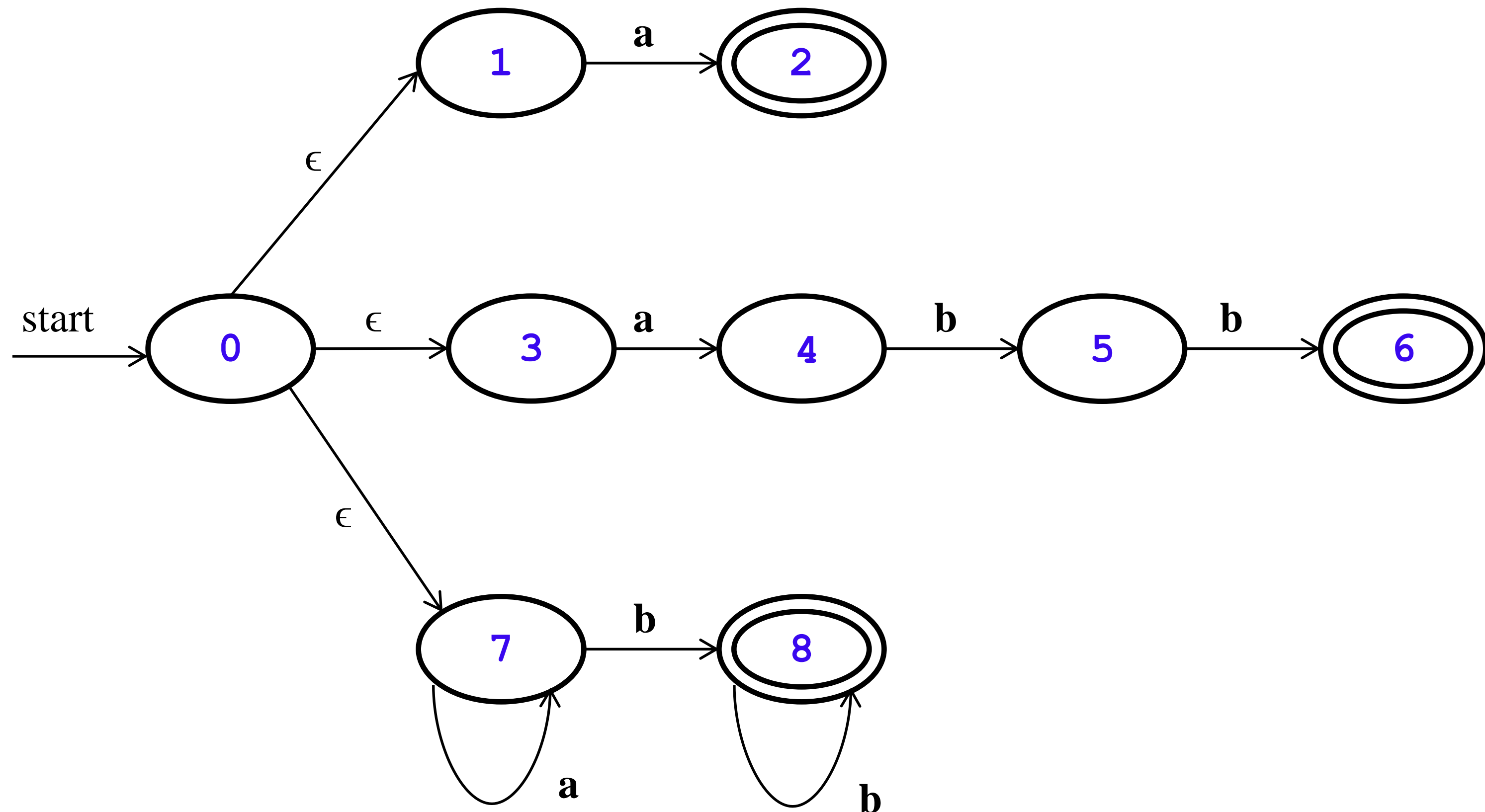

Figure 2: **Combined NFA**

Pattern Matching Based on NFA's

- If the lexical analyzer simulates an **NFA** such as that of ***Fig. 2***, 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.
- Eventually, the **NFA** simulation reaches a point on the input where there are no next states.

Pattern Matching Based on NFA's

- At that point, there is no hope that any longer prefix of the input would ever get the **NFA** to an accepting state; rather, the set of states will always be empty.
- Thus, we are ready to decide on the longest prefix that is a lexeme matching some pattern.

Pattern Matching Based on NFA's

- We look backwards in the sequence of sets of states, until we find a set that includes one or more accepting states.
- If there are several accepting states in that set, pick the one associated with the earliest pattern p_i in the list from the *Lex* program.
- Move the *forward* pointer back to the end of the lexeme, and perform the action A_i associated with pattern p_i.

Pattern Matching Based on NFA's

Example:

Suppose we have the patterns

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

and the input begins **aaba**.

Pattern Matching Based on NFA's

Example:

The following ***Figure 3*** shows the sets of states of the **NFA** of ***Fig. 2*** that we enter, starting with **ϵ-closure** of the initial state **0**, which is **{0,1,3,7}**, and proceeding from there.

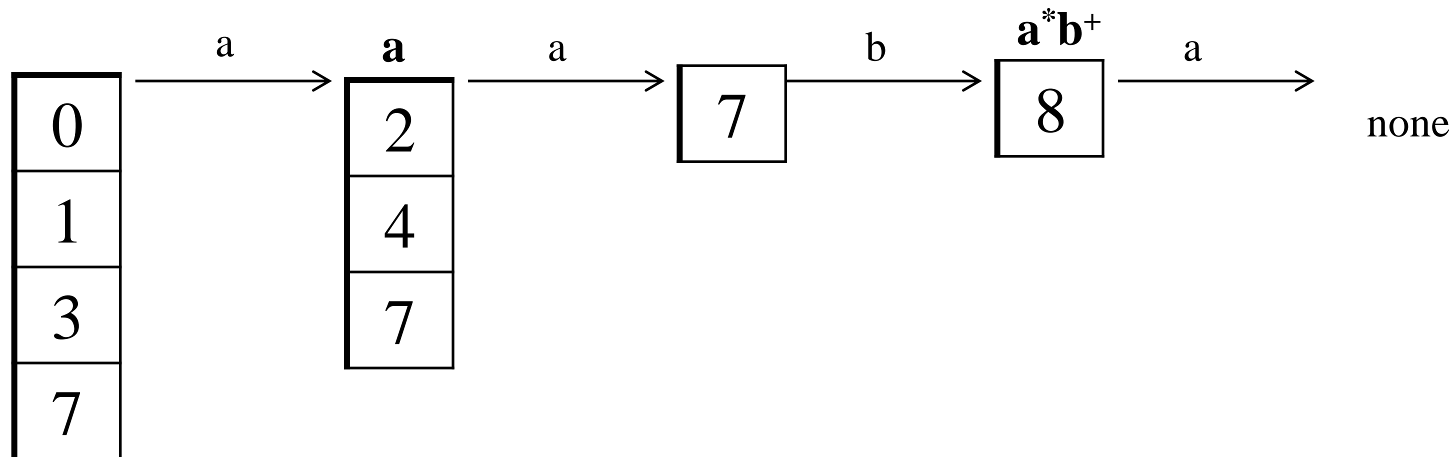

Figure 3: Sequence of set of states entered when processing input **aaba**

DFA's for Lexical Analyzers

- Another architecture, resembling the output of **Lex**, is to convert the **NFA** for all the patterns into an equivalent **DFA**, using the **subset construction**. Within each **DFA** state, if there are one or more accepting **NFA** states, determine the first pattern whose accepting state is represented, and make that pattern the output of the **DFA** state.

DFA's for Lexical Analyzers

Example:

- The following ***Figure 4*** shows a transition diagram based on the **DFA** that is constructed by the subset construction from the **NFA** in ***Fig. 2***.
- The accepting states are labeled by the pattern that is identified by that state.
- For instance, the state **{6, 8}** has two accepting states, corresponding to patterns **abb** and **a^*b^+**.
- Since the former is listed first, that is the pattern associated with state **{6, 8}**.

DFA's for Lexical Analyzers

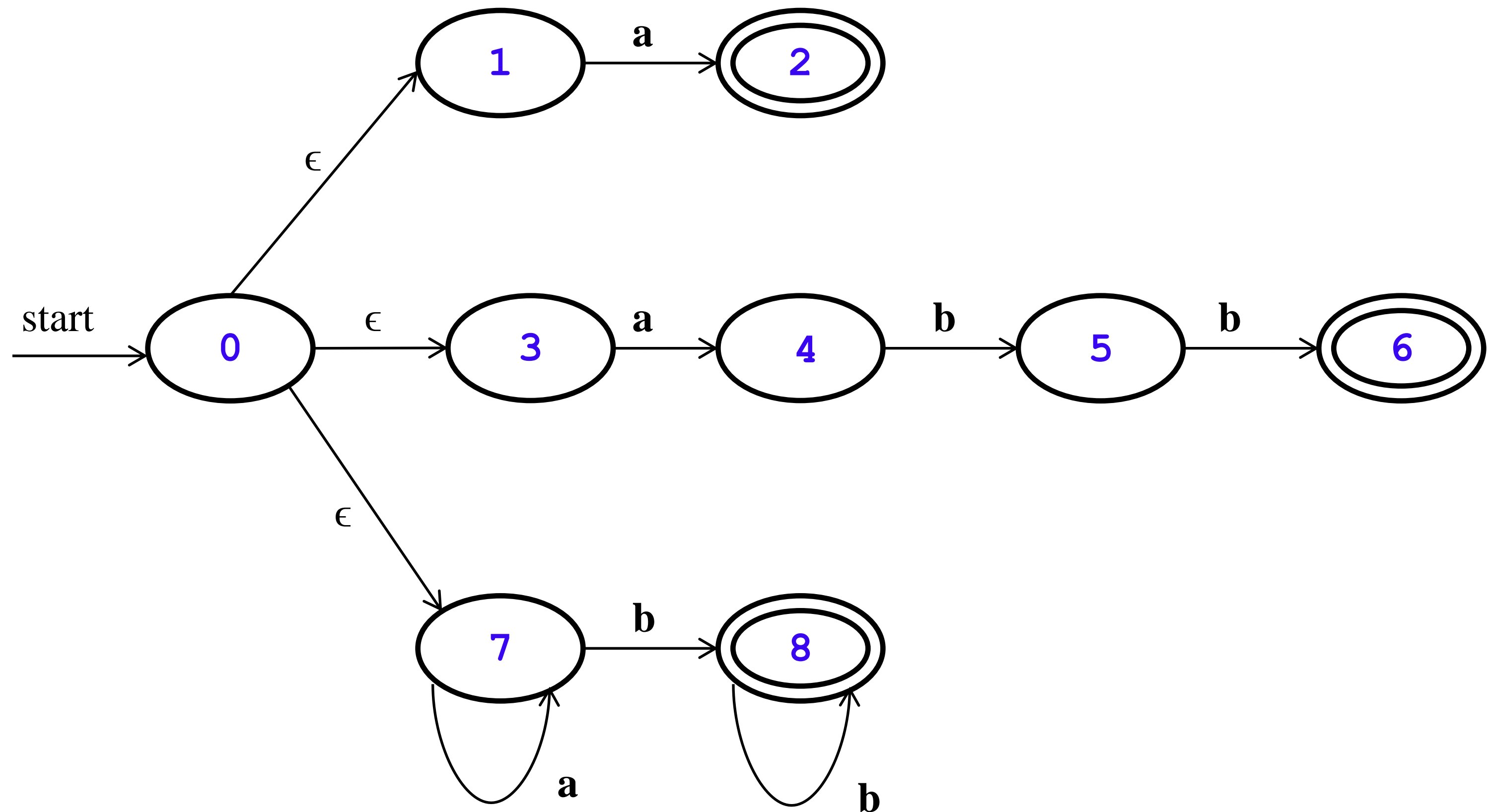

Figure 2: **Combined NFA**

DFA's for Lexical Analyzers

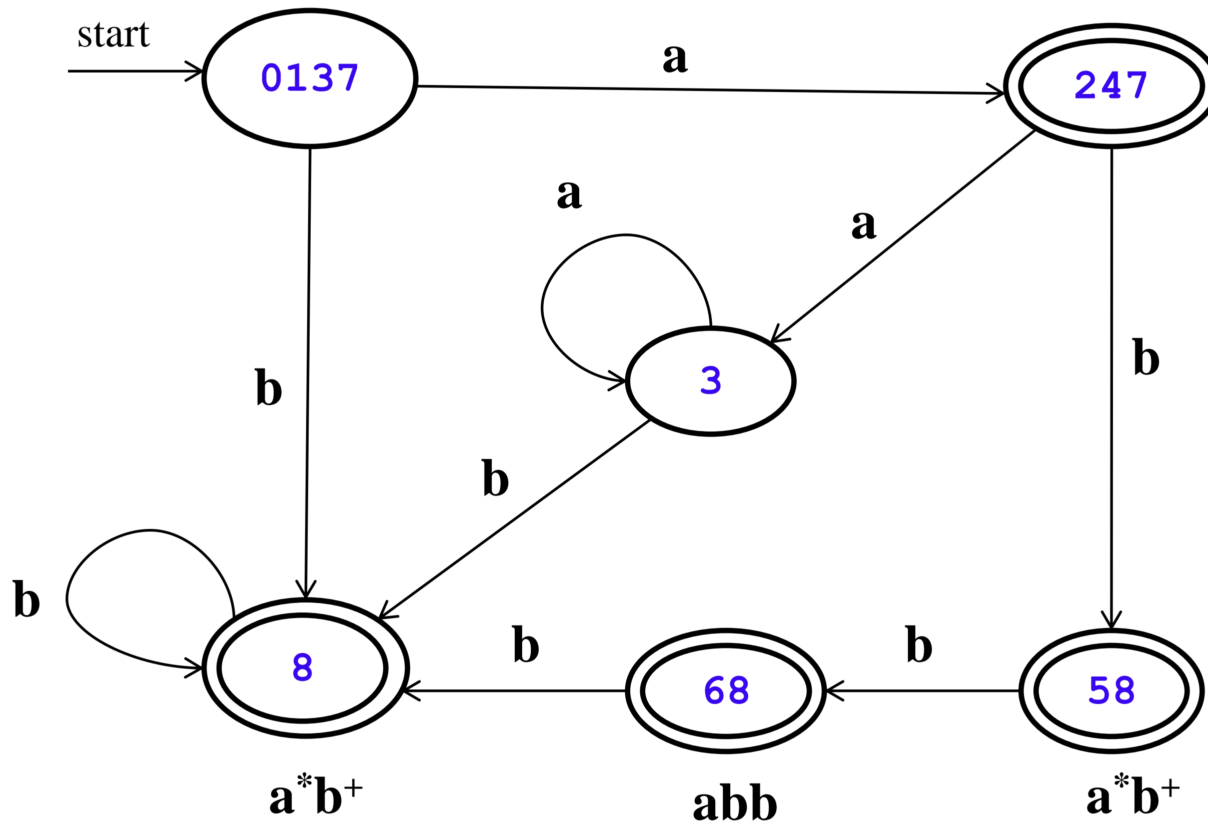

Figure 4: Transition graph for **DFA** handling the patterns **a**, **abb**, and **a*b+**

Implementing the Lookahead Operator

- The **Lex** *lookahead operator* **/** in a Lex pattern r_1/r_2 is sometimes necessary, because the pattern r_1 for a particular token may need to describe some trailing context r_2 in order to correctly identify the actual lexeme.
- When converting the pattern r_1/r_2 to an **NFA**, we treat the **/** as if it were ϵ, so we do not actually look for a **/** on the input.

Implementing the Lookahead Operator

- However, if the **NFA** recognizes a prefix **xy** of the input buffer as matching this regular expression, the end of the lexeme is not where the **NFA** entered its accepting state.
- Rather the end occurs when the **NFA** enters a state **s** such that
  1. **s** has an **ϵ-transition** on the (imaginary) **/**,
  2. There is a path from the start state of the **NFA** to state **s** that spells out **x**.
  3. There is a path from state **s** to the accepting state that spells out **y**.
  4. **x** is as long as possible for any **xy** satisfying conditions **1-3**.

Implementing the Lookahead Operator

- If there is only one **ϵ-transition** state on the imaginary **/** in the **NFA**, then the end of the lexeme occurs when this state is entered for the last time as the following example illustrates.
- If the **NFA** has more than one **ϵ-transition** state on the imaginary **/,** then the general problem of finding the correct state **s** is much more difficult.

Implementing the Lookahead Operator

Example:

An **NFA** for the pattern for the Fortran **IF** with lookahead, is shown in ***Fig. 5***.

Notice that the **ϵ-transition** from state **2** to state **3** represents the lookahead operator. State **6** indicates the presence of the keyword **IF**. However, we find the lexeme **IF** by scanning backwards to the last occurrence of state **2**, whenever state **6** is entered.

Implementing the Lookahead Operator

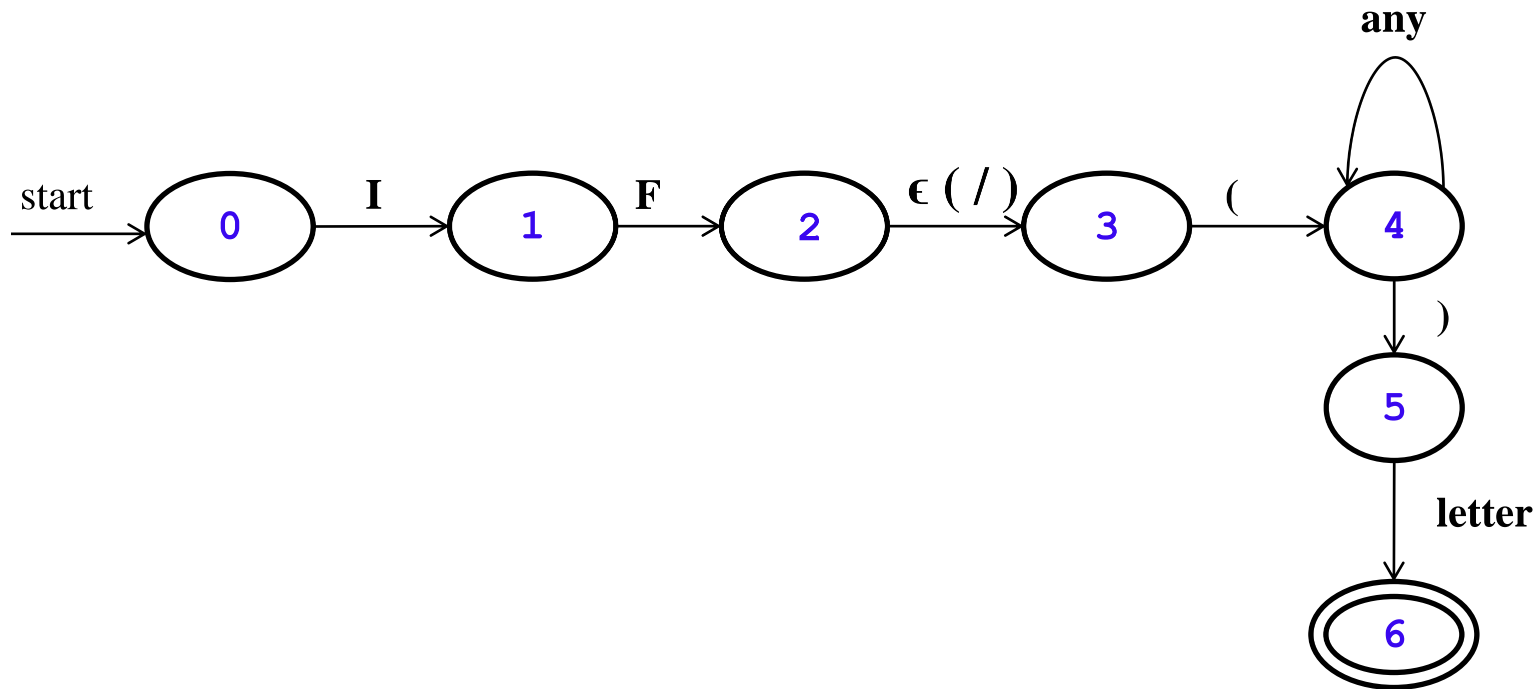

Figure 5: **NFA** recognizing the keyword **IF**