CS321 Languages and Compiler Design I

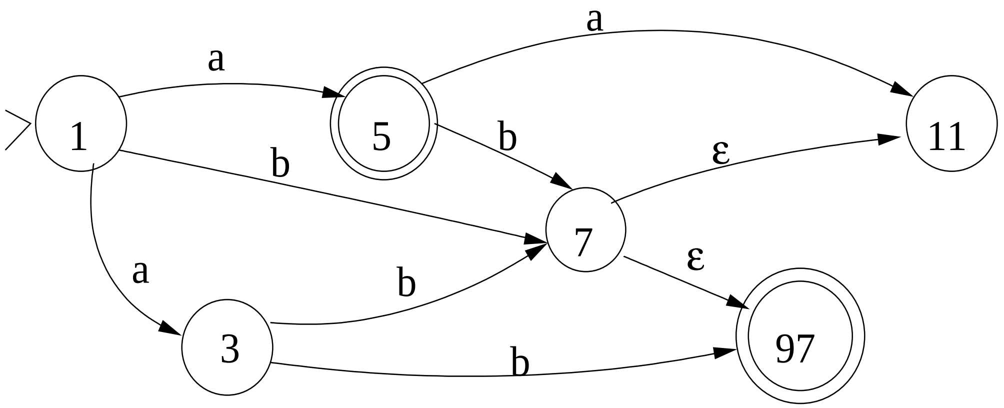
Winter 2012

Lecture 5

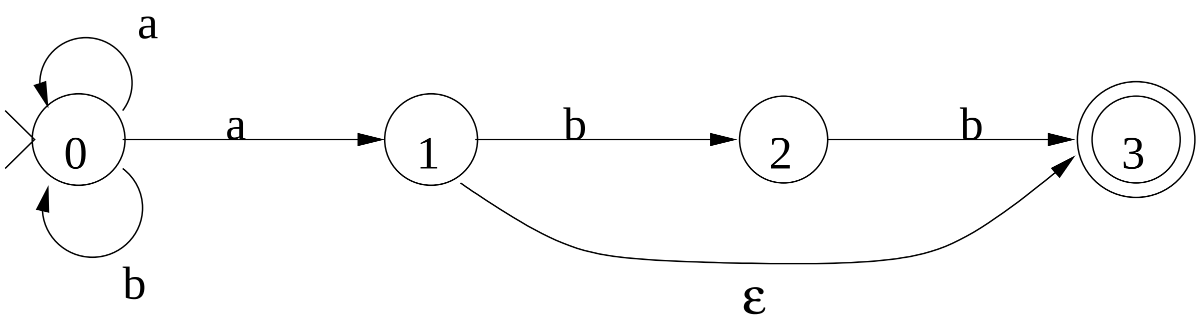
FINITE AUTOMATA

A non-deterministic finite automaton (NFA) consists of:

- An input alphabet Σ, e.g. $\Sigma = \mathrm{a}, \mathrm{b}$.
- A set of states S, e.g. $S = \{1, 3, 5, 7, 11, 97\}$.
- A designated start state, e.g. state 1.
- A designated set of final states, e.g. $\{5, 97\}$.
- A set of transitions from states to states, labelled by elements of Σ or ϵ , e.g.



NFA'S ACCEPT STRINGS



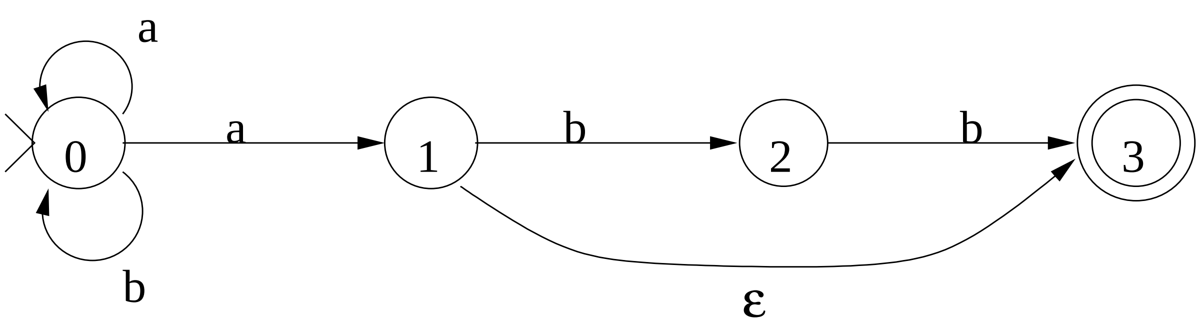
Can also write as **transition table**.

State	Input a	Input b	Input ϵ
$\rightarrow$ 0	{0,1}	{0}	-
1	-	{2}	{3}
2	-	{3}	-
$*$3	-	-	-

An NFA **accepts** the **string** x if there is a path from start to final state labeled by the the characters of x, possibly including some ϵ 's.

Example: NFA above accepts the strings "aaabb", "aaabbabb", and "a", among many others.

NFA's and Regular Languages



An NFA **accepts** the **language** L if it accepts exactly the strings in L.

Example: NFA above accepts the language defined by the R.E.
$(\mathrm{a}^*\mathrm{b}^*)^*\mathrm{a}(\mathrm{bb} \mid \epsilon\ )$.

Fact: For every regular language L, there exists an NFA that accepts L.

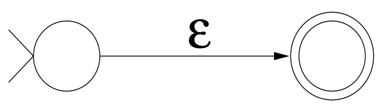
NFA's FROM R.E.'s

Can give an algorithm for constructing an NFA from an R.E., such that the NFA accepts the language defined by the R.E.

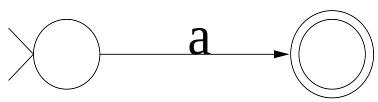
- Algorithm is recursive, and is based on the recursively defined structure of R.E.'s.
- Makes heavy use of ϵ -transitions.

Base Constructions

ε



a in Σ

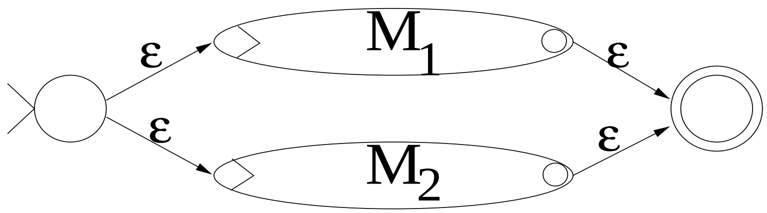


Inductive Constructions build new machines by connecting existing machines using ϵ -transitions to existing initial states and from existing final states.

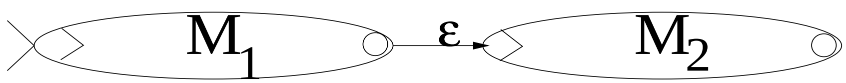
Note that each constructed machine has exactly one initial state and one final state.

Inductive Constructions

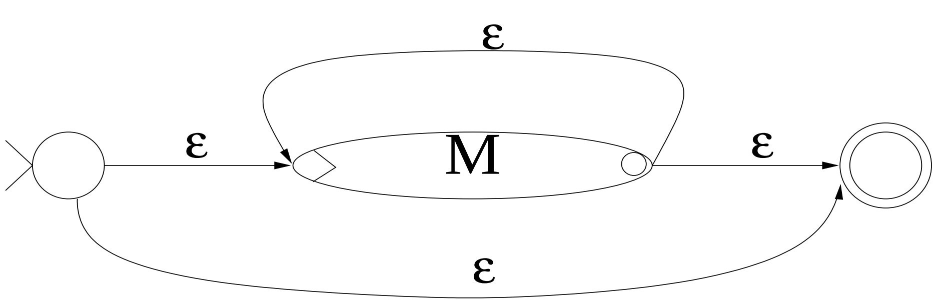
$R_1 | R_2$



$R_1 \cdot R_2$



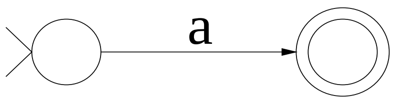
R^*



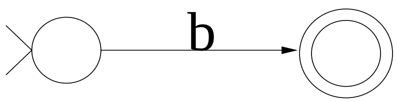
NFA CONSTRUCTION EXAMPLE

Example: (a|b)*|(cbb)

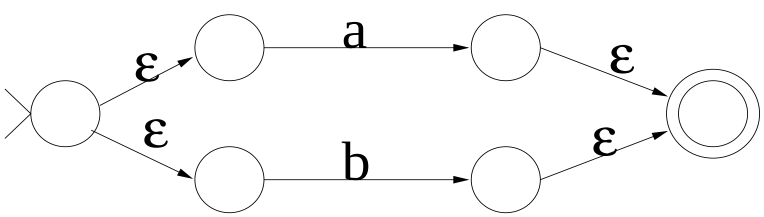
a



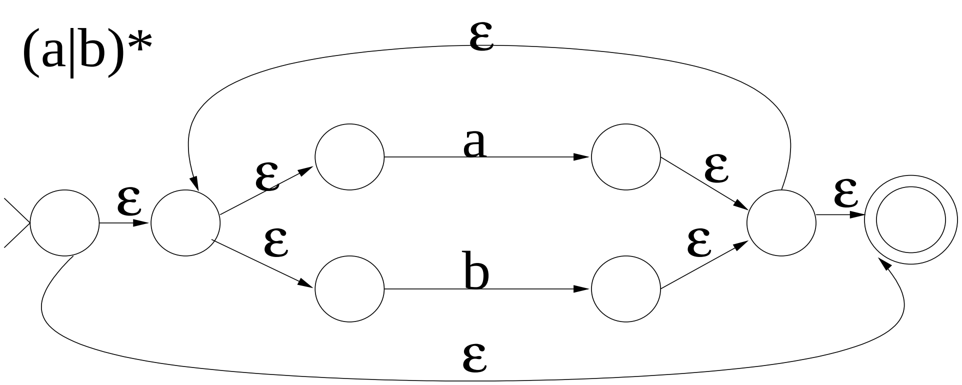
b



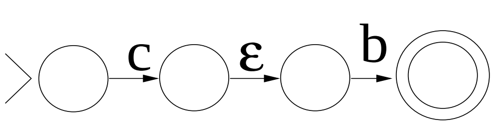
a|b



(a|b)*

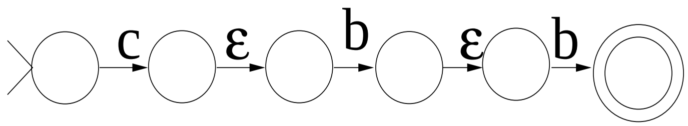


cb

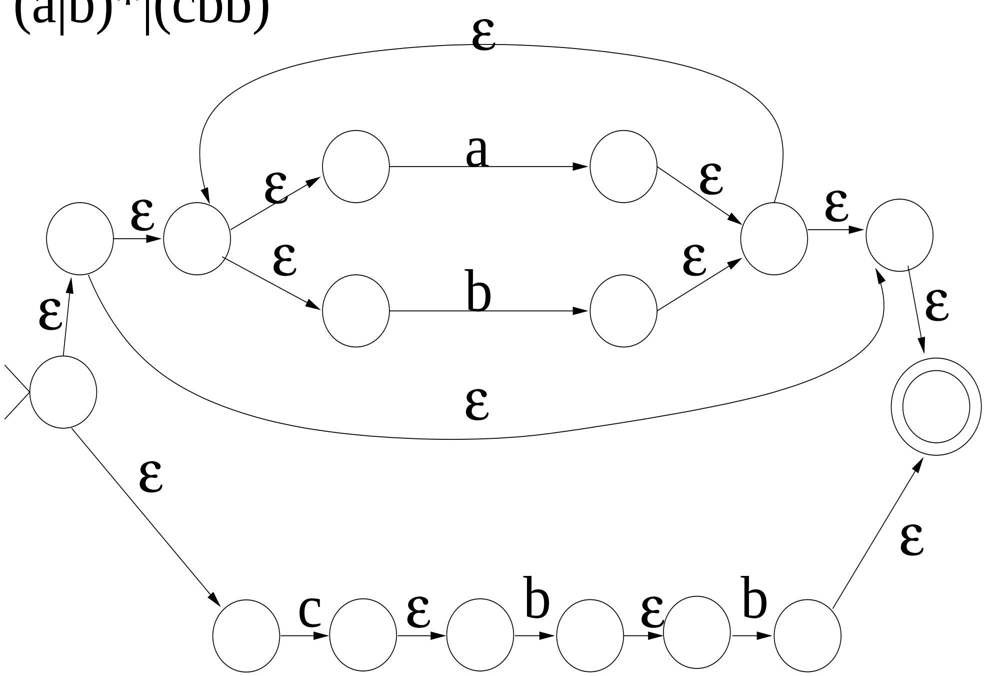


Example (continued)

cbb



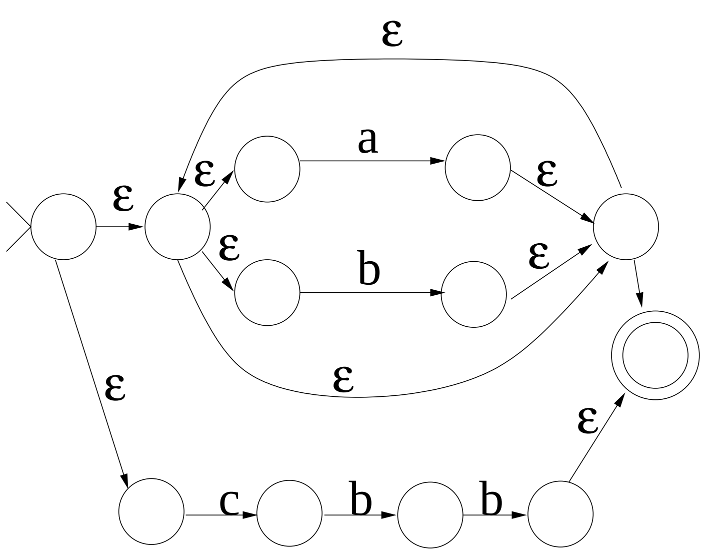
(a|b)*|(cbb)



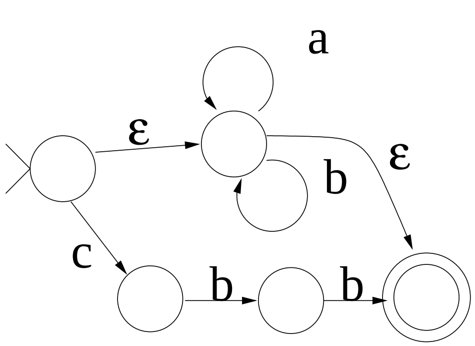
Example (continued)

Can simplify NFA's by removing useless empty-string transitions:

(a|b)*|(cbb)

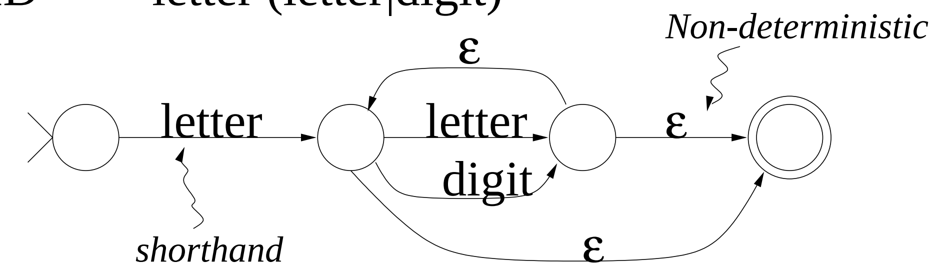


Or even simpler:

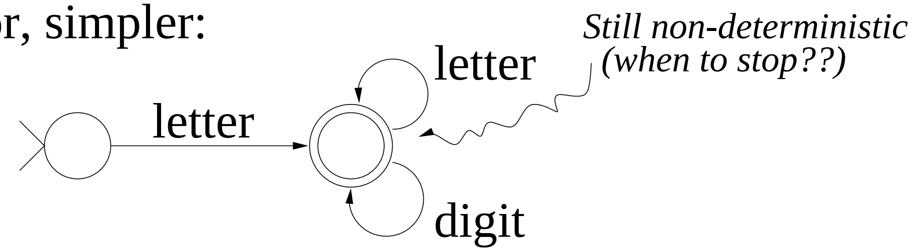


NFA's for Lexical Pattern R.E.s'

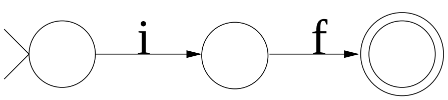
ID letter (letter|digit)*



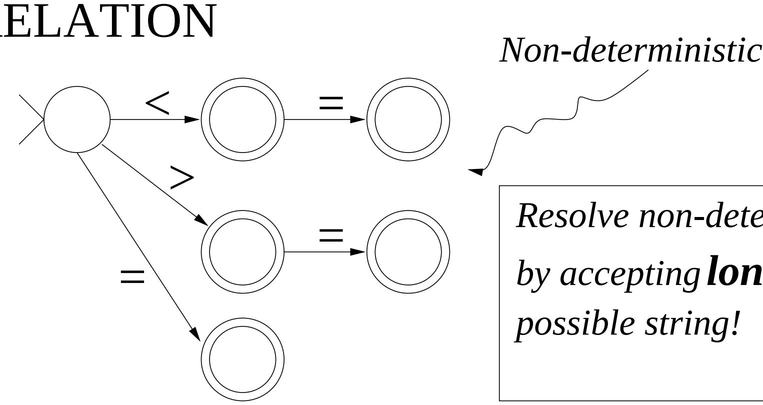
or, simpler:



IF



RELATION



Resolve non-determinism by accepting ***longest*** *possible string!*

Lexical analyzer must find **best** match among a **set** of patterns.

Try: NFA for pattern #1

Then try: NFA for pattern #2

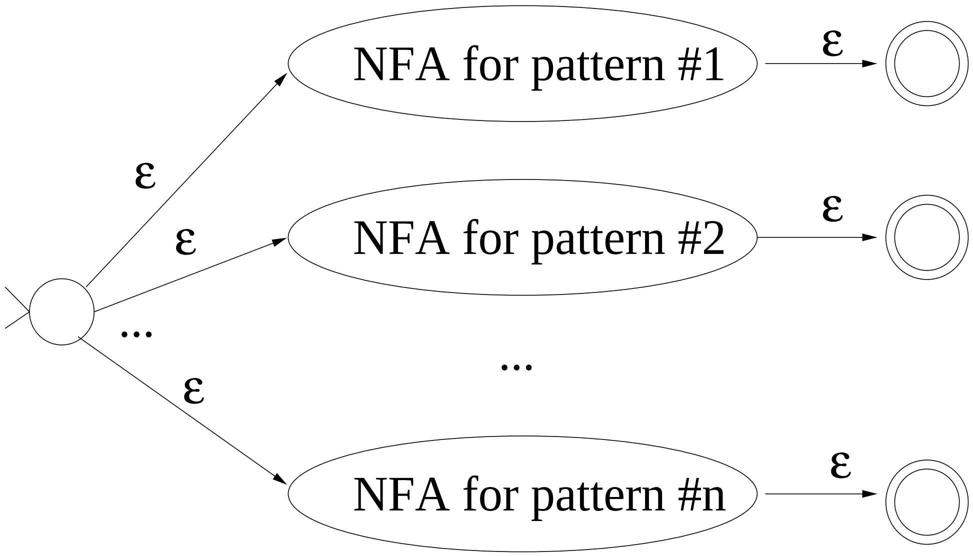
...

Finally, try: NFA for pattern #n

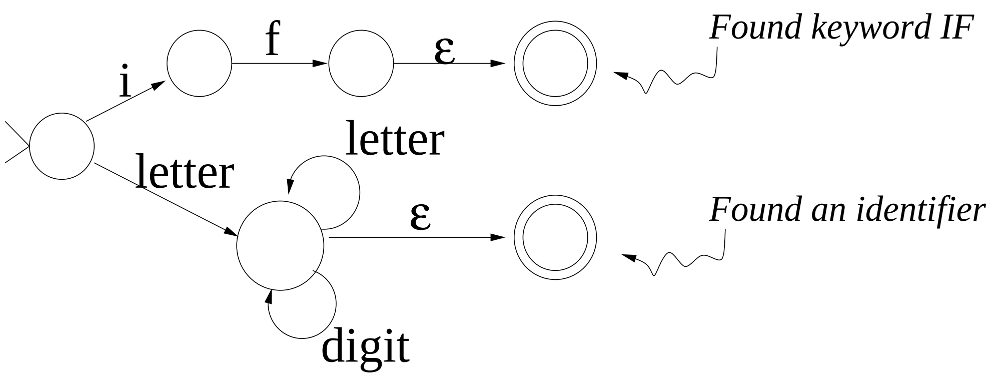
Must reset input string after each unsuccessful match attempt.

Always choose pattern that allows longest input string to match. Must specify which pattern should 'win' if two or more match the same length of input.

Alternatively, combine all the NFA's into one giant NFA, with distinguished final states:



Now can have non-determinism between patterns, as well as within a single pattern, e.g:



IMPLEMENTING NFA'S

Behavior of an NFA on a given input string is ambiguous.

So NFA's don't lead to a deterministic computer program.

Can convert to **deterministic** finite automaton (DFA).

• (Also called "finite state machine.")

• Like NFA, but has no ϵ -transitions and no symbol labels more than one transition from any given node.

• Easy to simulate on computer.

• There is an algorithm ("subset construction") that can convert **any** NFA to a DFA that accepts the same language.

Alternative approach: Simulate NFA directly by pretending to follow all possible paths "at once."

To handle "longest match" requirement, must keep track of last final state entered, and backtrack to that state ("unreading" characters) if get stuck.

DFA and Backtracking Example

Given the following set of patterns:

a

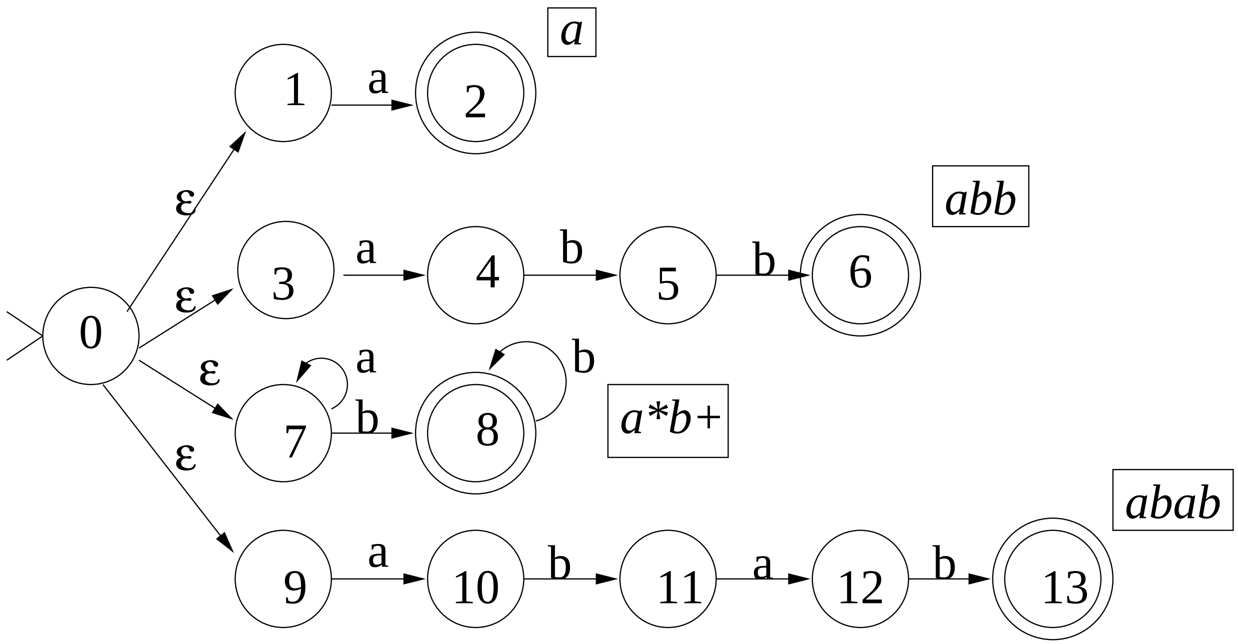
abb

a^*b^+

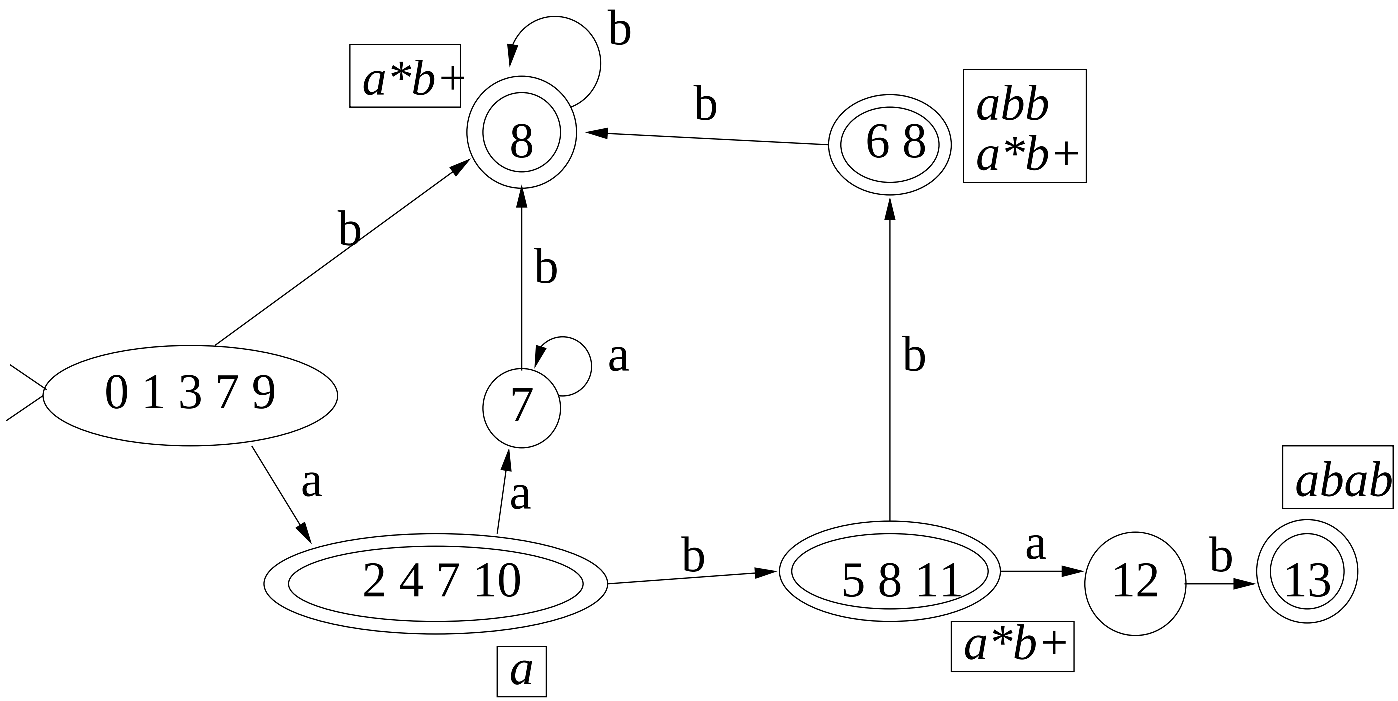
abab

We want to build a machine to find the longest match; in case of ties, favor the pattern listed first.

Here's the NFA:



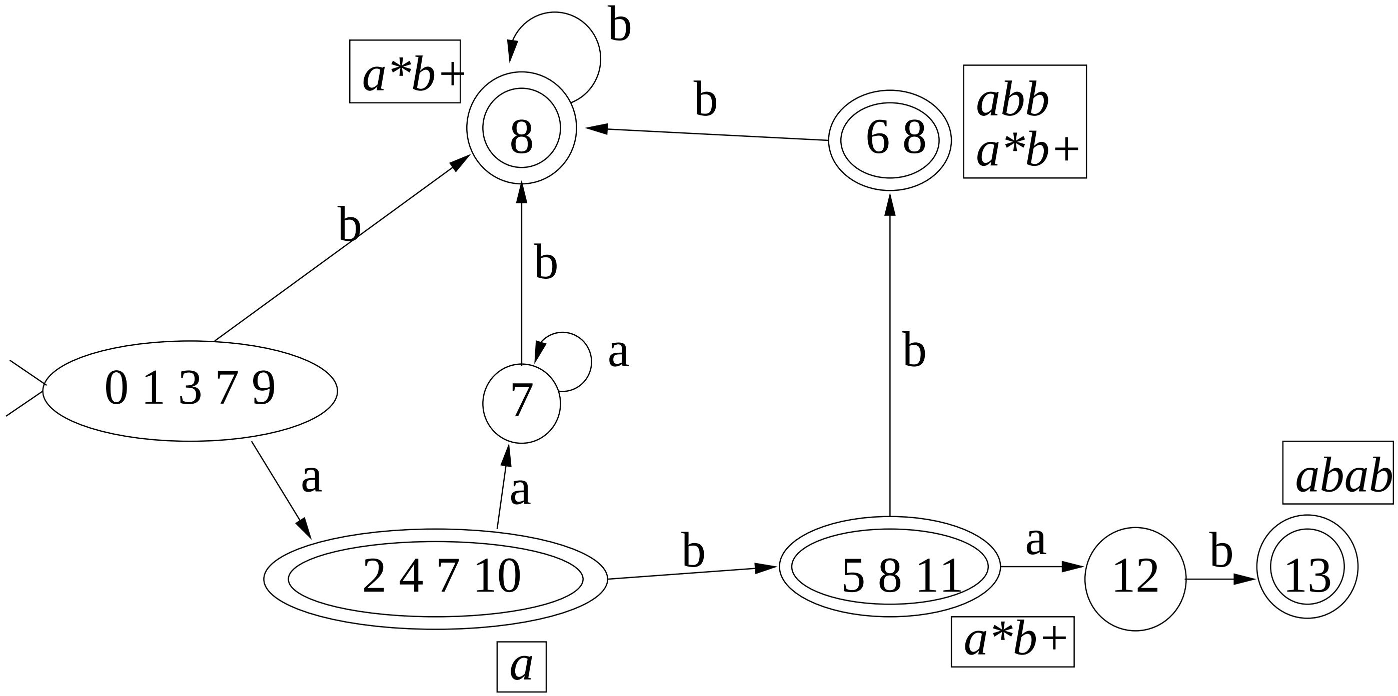
CORRESPONDING DFA



Consider input “a” :

- Machine stops in state (2 4 7 10).
- Pattern is a.
- Lexeme is “a”.

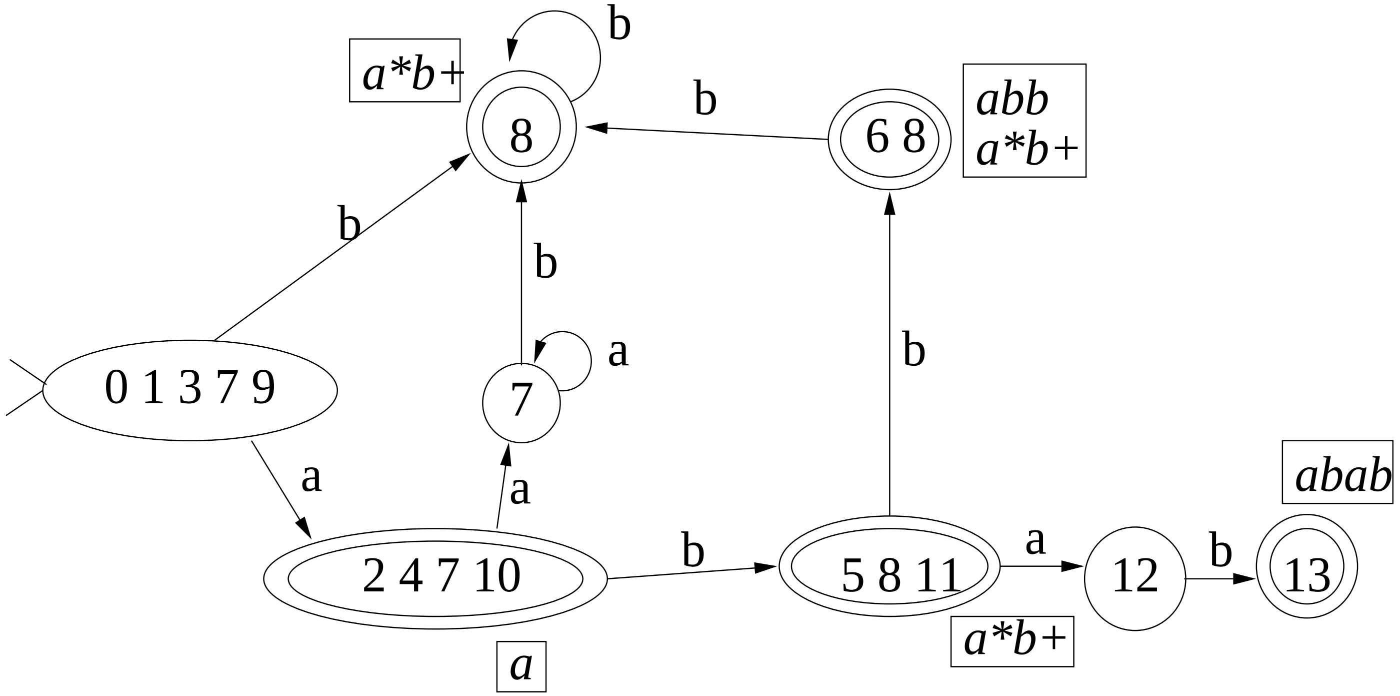
CORRESPONDING DFA



Conider input "aaab":

- Machine stops in state (8)
- Pattern is $a^{*}b+$.
- Lexeme is "aaab".

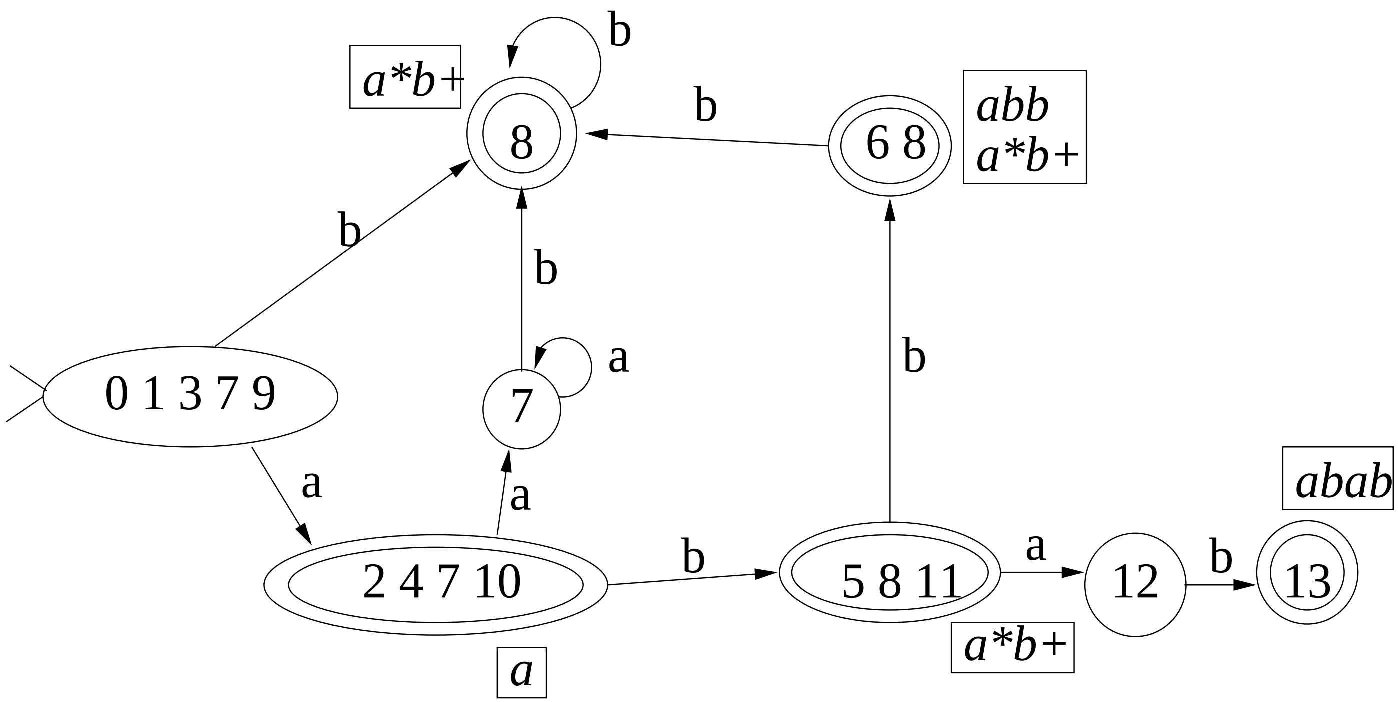
CORRESPONDING DFA



Consider input "abba" :

- Machine stops after second "b" in state (6 8).
- Pattern is abb because it comes first in spec.
- Lexeme is "abb"; final "a" will be read again next time.

CORRESPONDING DFA



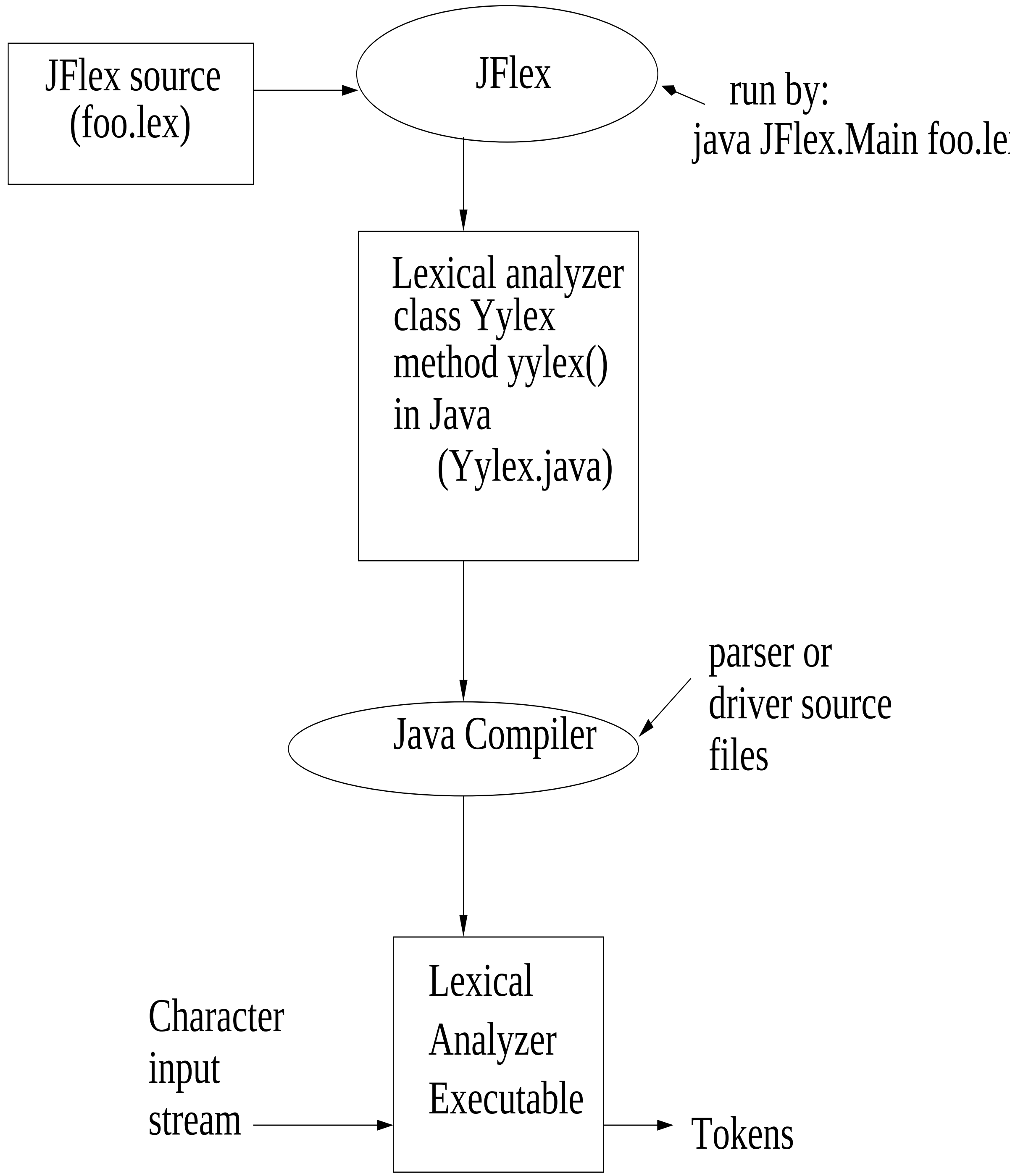
Consider input “abaa”:

- Machine gets stuck after “aba” in state (12).
- Backs up to state (5 8 11), unreading “a”
- Pattern is a^*b^+
- Lexeme is “ab”; final “aa” will be read again next time.

JFLEX

`JFlex` is a **lexical analyzer generator**

- Java version of original AT&T `lex` tool for C; many similar tools exist. Details of use may vary.
- accepts **specification** of lexical analyzer.
- produces Java program that **implements** specification.


JFlex source
(foo.lex)
JFlex
run by:
java JFlex.Main foo.le
Lexical analyzer
class Yylex
method yylex()
in Java
(Yylex.java)
parser or
driver source
files
Java Compiler
Character
input
stream
Lexical
Analyzer
Executable
Tokens

JFLEX RULE SPECIFICATIONS

The main input to `JFlex` is a sequence of **rules**, each consisting of a

- Pattern – regular expression (using ASCII as alphabet)
- Action – fragment of Java code

When prefix of input matches a pattern, the generated analyzer executes the corresponding action.

Actions can make use of built-in variables and methods

- `yytext()` returns lexeme as a `String`
- `yyline` contains current line number (must use `%line` option).

JFlex Rules Example

```
%%
%%
integer        {println("found keyword INTEGER");}
[0-9]+         {println("found number");}
[A-Z][A-Z]*    {println("found ident " + yytext());}
[ \t\n]        { /* ignore white space */ }
```

As usual, if more than one pattern matches, the longest match is preferred; ties are broken in favor of rule that appears first.

JFLEX PATTERNS

Patterns include literal text and meta-level operators.

Pattern	Matches
`x`	character "`x`"
`"x"`	character "`x`" even if it's an operator
`\x`	ditto
`[xy]`	"`x`" or "`y`"
`[x-y]`	characters between "`x`" and "`y`" inclusive
`[^s]`	any character not in set s
`.`	any character but "`\n`"
p?	an optional p
p*	zero or more p's
p+	one or more p's
p|q	p or q
()	grouping
{d}	substitute **definition** for d

JFLEX ACTIONS

Actions can be any valid Java statement block.

Ordinarily each action terminates with a statement `return` t`;` which causes `yylex()` to return with the token value t.

Otherwise, `yylex()` throws away the lexeme and continues searching for another pattern. This is suitable for handling white space. The simplest possible action is just the empty block $\{\}$.

`yylex()` raises an exception if no pattern matches. So it is a good idea to include a “catch-all” pattern as the last rule, e.g.:

```
. { System.err.println("Unexpected character"); }
```

JFLEX DIRECTIVES

The complete form of a `JFlex` specification is:

user code

`%%`

JFlex directives

`%%`

rules

Directives include control instructions, such as `%line`, which says the generated code should keep track of line numbers.

Directives can also include macro **definitions**, which abbreviate regular expressions for later use in patterns, e.g.,

```
%%
LETTERS=[a-zA-Z_]
DIGITS=[0-9]
%%
{LETTERS}({LETTERS}|{DIGITS})* {return new Token(ID);}
```

JFLEX USER CODE

User code is just copied directly to the top of the generated `.java` file; it can contain functions and globals to be invoked from the actions.

Such code can also be included in the directives section if enclosed between `%{` and `%}`; in this case, it is copied into the *inside* of the generated `Yylex` class.

JFlex States

`JFlex` permits multiple sets of rules to coexist in the same specification. Each set of rules is associated with a **state**.

Rules prefixed with <*name*> are recognized (only) when `yylex()` is in the state *name*.

When `yylex()` starts running it is in the state with the predefined name `YYINITIAL`.

You declare new state names in a `%state` line in the definitions section of the spec. You put `yylex()` into state *name* by including the method call

`yybegin(`*name*`);`

in an action.

EXAMPLE USING JFLEX STATES

Example: multi-line comments in Java.

```
%%
%state COMMENT
%%
<YYINITIAL>"/*"   { yybegin(COMMENT); }
<COMMENT>"*/"     { yybegin(YYINITIAL); }
<COMMENT>.|"\n"   { /* ignore comments */ }
<YYINITIAL> ...    ordinary rules follow
<YYINITIAL> ...
```