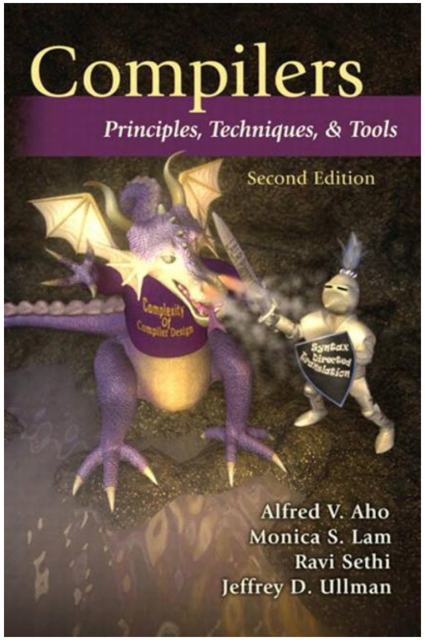
Compiler

Lec 03

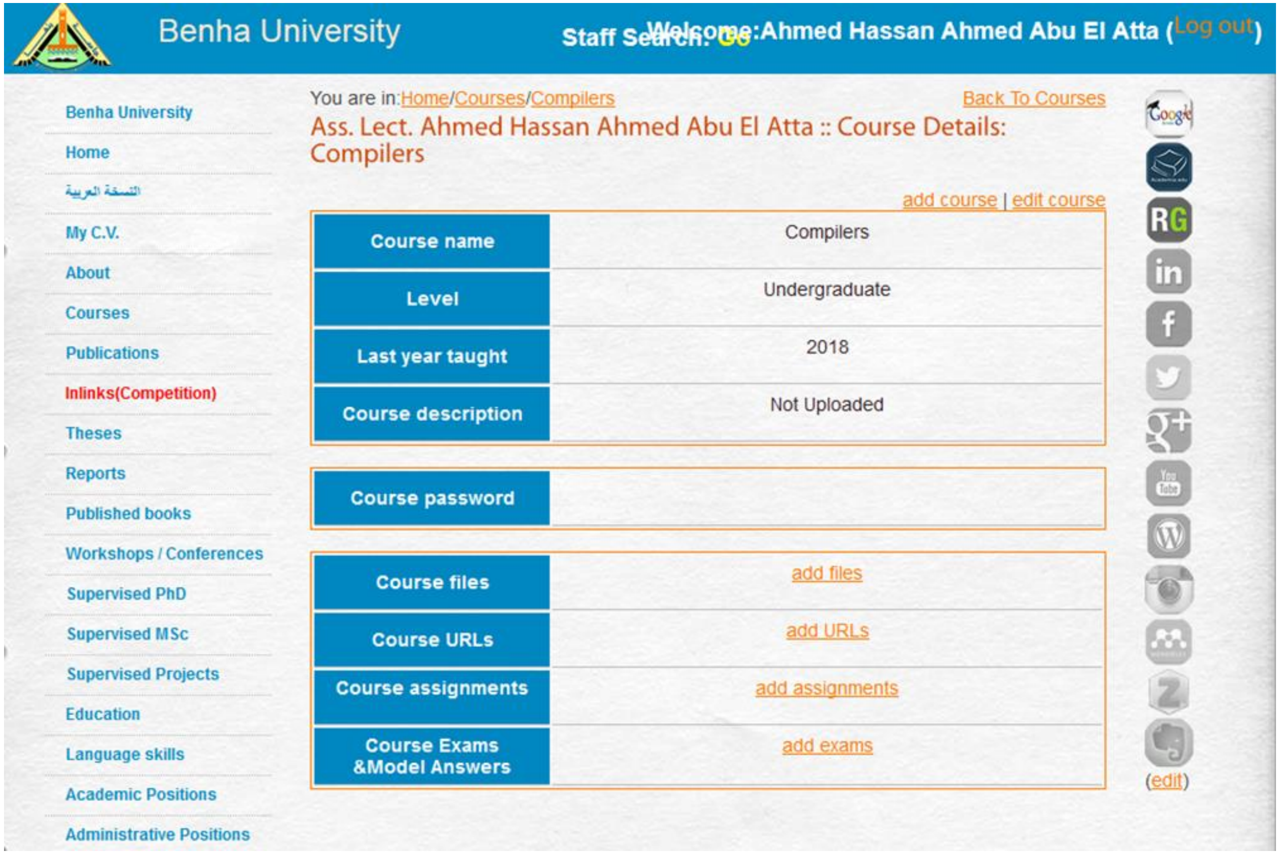
Book

Compilers: Principles, Techniques, and Tools is a computer science textbook by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman about compiler construction.



PowerPoint

http://www.bu.edu.eg/staff/ahmedaboalatah14-courses/14779



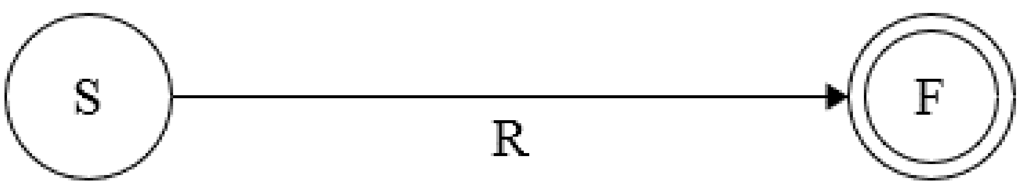
Lexical Analysis

PART TWO

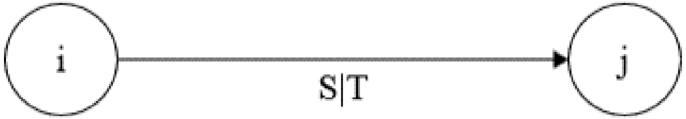
RE to NFA

Start with two states; one is the start state and the another is the final state.

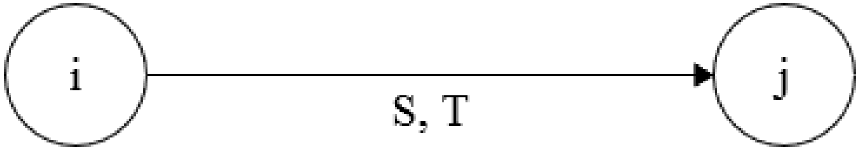
Connect them by one edge labeled with the regular expression.



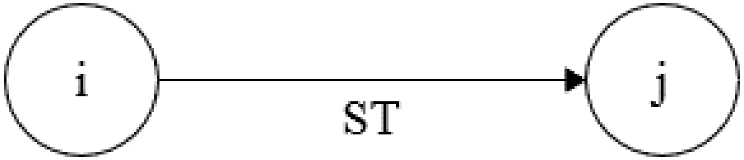
RE to NFA (cont.)



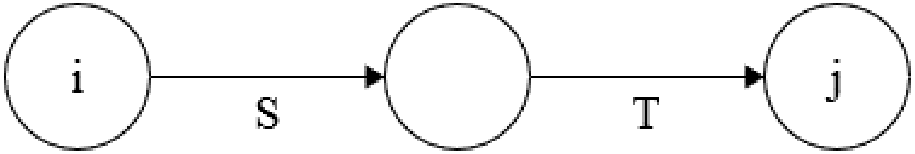
For the regular expression S|T,



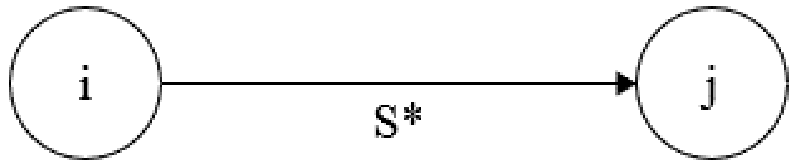
RE to NFA (cont.)



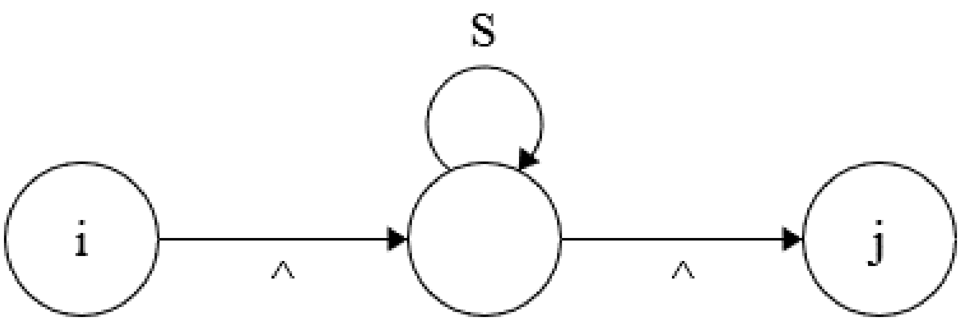
For the regular expression ST,



RE to NFA (cont.)



For the regular expression *s**,



Examples

- Convert the following RE to NFA:
  - a | b
  - (a | b) (a | b)
  - a *
  - (a | b)*
  - a | a*b

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) consists of:

- A finite set of states S.
- A set of input symbols ∑, the input alphabet. We assume that Ɛ, which stands for the empty string, is never a member of ∑ .
- A transition function that gives, for each state, and for each symbol in ∑ U {Ɛ} a set of next states.
- A state s_0 from S that is distinguished as the start state (or initial state) .
- A set of states F, a subset of S, that is distinguished as the accepting states (or final states) .

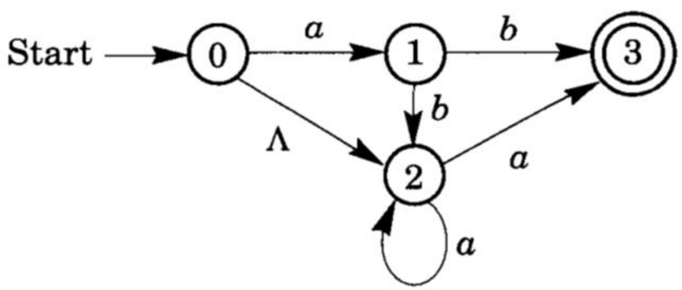
Deterministic Finite Automata

A deterministic finite automaton (DFA) is a special case of an NFA where:

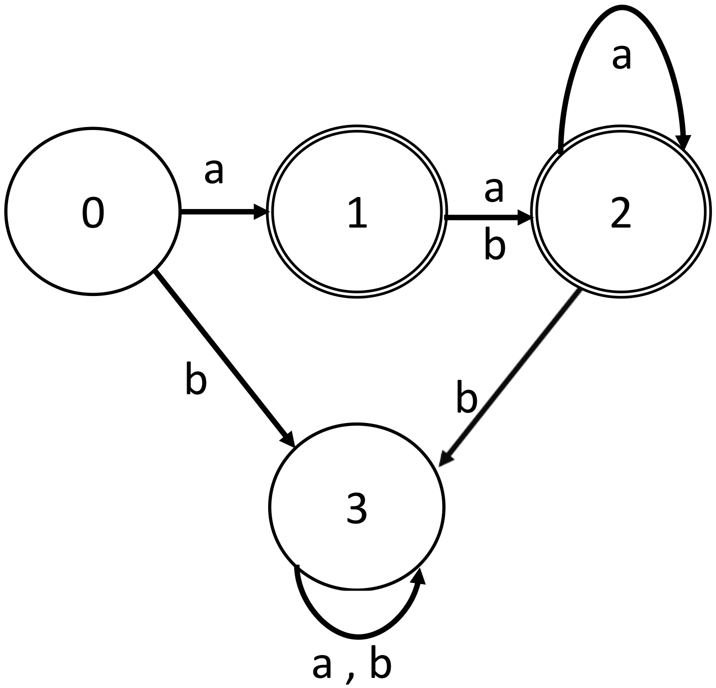
- There are no moves on input "Ɛ", and
- For each state "s" and input symbol "a", there is exactly one edge out of "s" labeled "a".

Example

NFA



DFA



a(Ɛ|b)a*

Simulating a DFA

INPUT:

- An input string “x” terminated by an end-of-file character “eof”.
- A DFA “D” with start state “s_0”, accepting states “F”, and transition function move.

OUTPUT:

- Answer "yes" if D accepts x ;
- "no" otherwise.

METHOD:

- The function *move*(s, c) gives the state to which there is an edge from state s on input c.
- The function *nextChar* returns the next character of the input string x.

Simulating a DFA Algorithm

$$
\begin{aligned}
&s = s_0; \\
&c = nextChar(); \\
&\textbf{while}\ (\ c\ != \textbf{eof}\ )\ \{ \\
&\qquad s = move(s, c); \\
&\qquad c = nextChar(); \\
&\} \\
&\textbf{if}\ (\ s \text{ is in } F\ )\ \textbf{return "yes";} \\
&\textbf{else return "no";}
\end{aligned}
$$

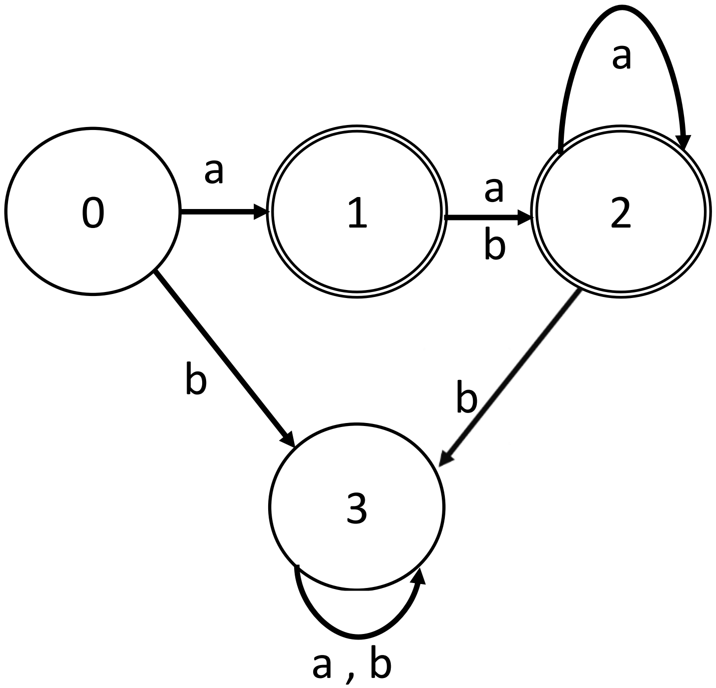
Example

X = ab

s =0 , c = a	1
s =1 , c = b	1
s =2 , c = eof	1
Yes	3

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

DFA



Simulating a NFA

INPUT:

- An input string “x” terminated by an end-of-file character “eof”.
- An NFA “N” with start state “s_0“, accepting states “F”, and transition function move.

OUTPUT:

- Answer "yes" if M accepts x ;
- "no" otherwise.

METHOD :

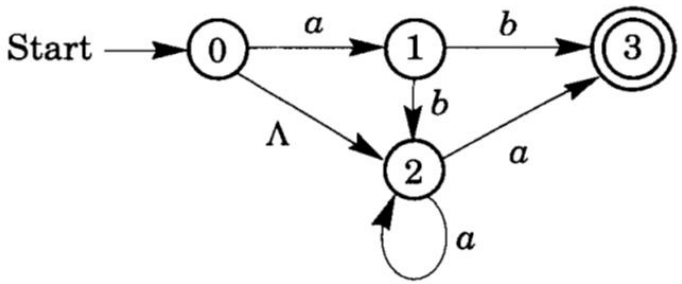
- The algorithm keeps a set of current states “S”, those that are reached from “s_0“ following a path labeled by the inputs read so far.
- If “c” is the next input character, read by the function *nextChar*() , then we first compute *move*(S, c) and
- then close that set using *Ɛ–closure*() .

Simulating a NFA Algorithm

```
S = ϵ-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ϵ-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

Example

NFA




```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

X = ab

S = {0, 2}, c = a	2
S = {1, 2, 3}, c = b	3
S = {2, 3}, c = eof	2
Yes	7

NFA to DFA

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

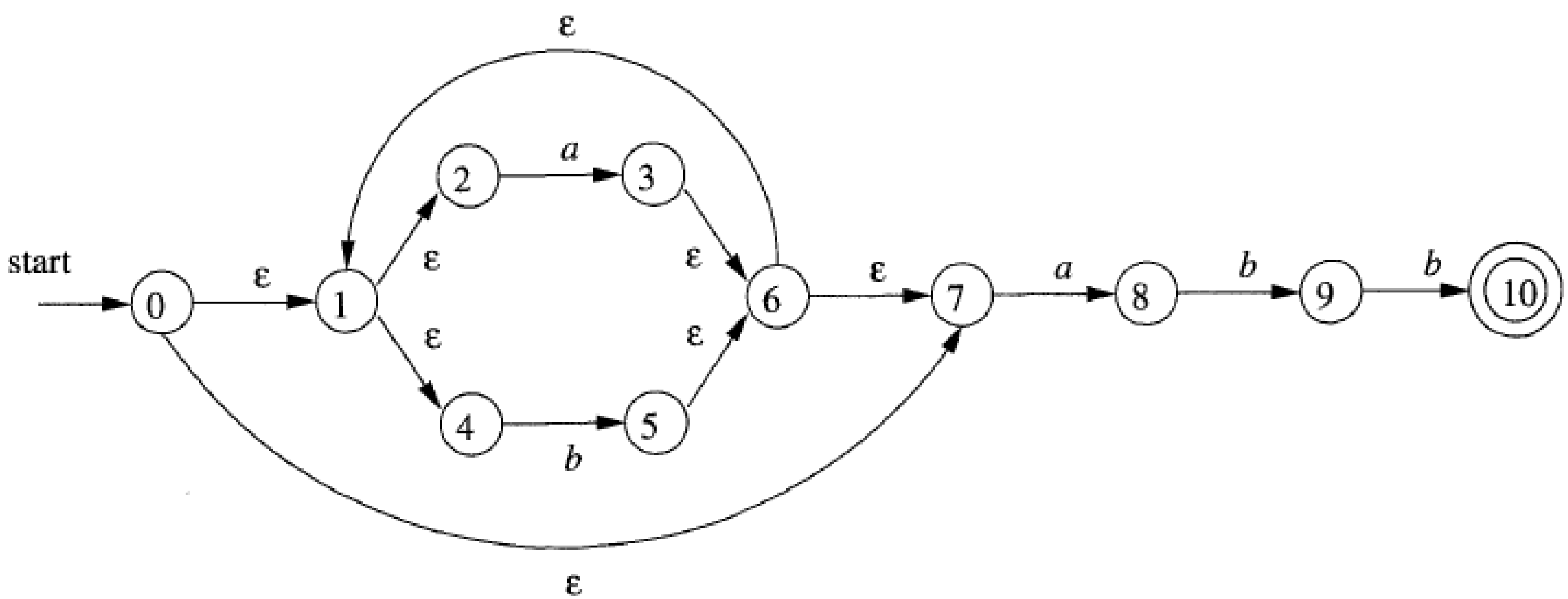
NFA to DFA (Computing Ɛ- closure(T))

```
push all states of T onto stack;
initialize ϵ-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ϵ )
                if ( u is not in ϵ-closure(T) ) {
                        add u to ϵ-closure(T);
                        push u onto stack;
                }
}
```

NFA to DFA
(The subset construction)

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

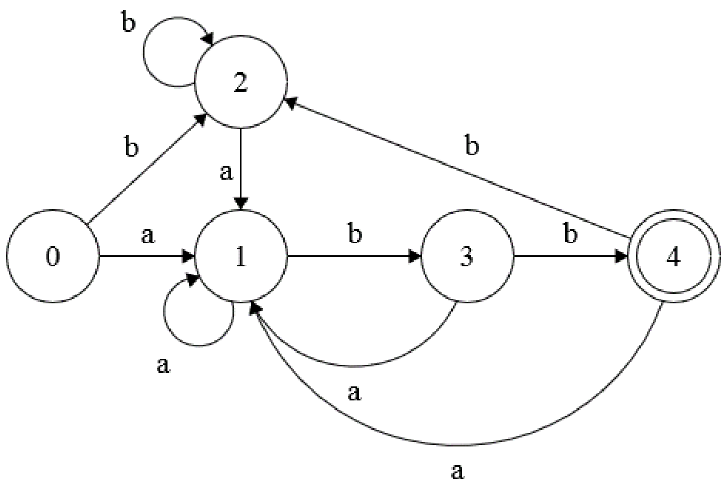
Example



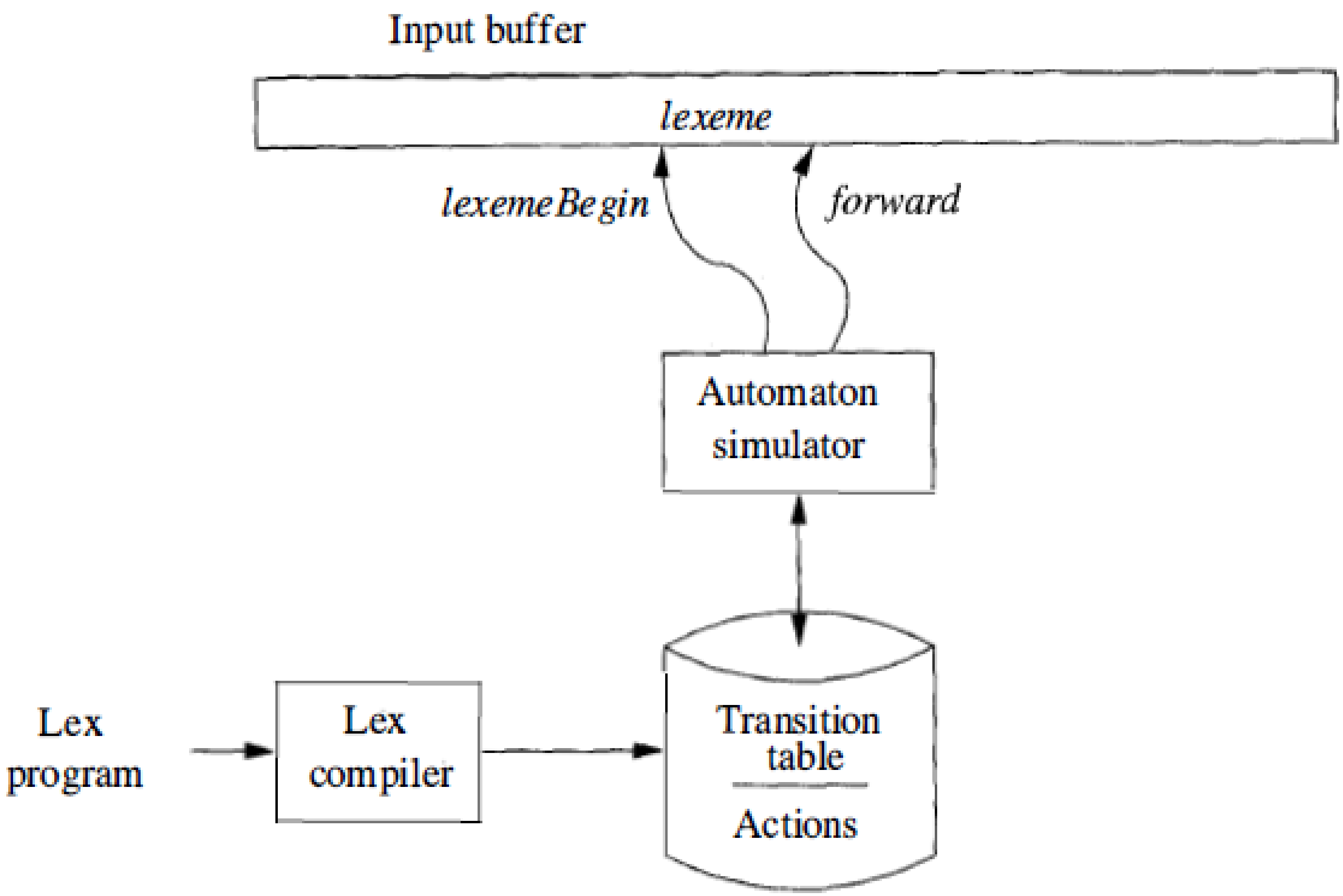
Example
Computing Ɛ- closure

state	a	b	Ɛ	Ɛ- closure
0	-	-	{1, 7}	{0, 1, 2, 4, 7}
1	-	-	{2, 4}	{1, 2, 4}
2	{3}	-	-	{2}
3	-	-	{6}	{1,2,3,4,6,7}
4	-	{5}	-	{4}
5	-	-	{6}	{1,2,4,5,6,7}
6	-	-	{1, 7}	{1,2,4,6,7}
7	{8}	-	-	{7}
8	-	{9}	-	{8}
9	-	{10}	-	{9}
10	-	-	-	{10}

NFA Set		a	b
{0,1,2,4,7}	***0***	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
{1,2,3,4,6,7,8}	***1***	{1,2,3,4,6,7,8}	{1,2,4,5,6,7,9}
{1,2,4,5,6,7}	***2***	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
{1,2,4,5,6,7,9}	***3***	{1,2,3,4,6,7,8}	{1,2,4,5,6,7,10}
{1,2,4,5,6,7,10}	***4***	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}



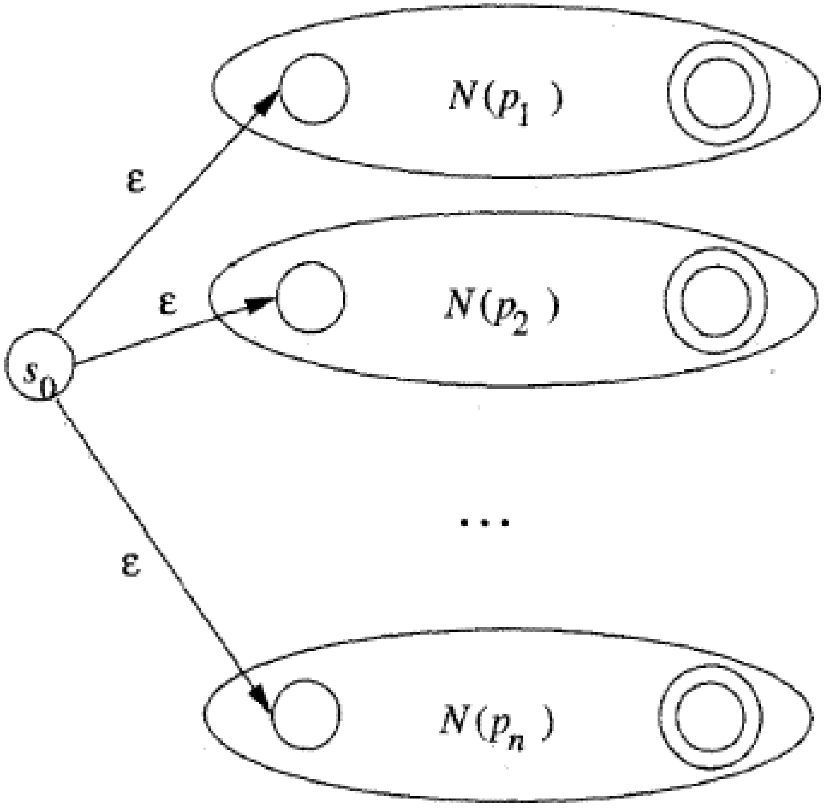
The Structure of the Generated Analyzer



Construct Scanner

To construct the automaton:

- We begin by taking each regular-expression pattern in the language and converting it to an NFA.
- We combine all the NFA's into one by introducing a new start state with Ɛ-transitions to each of the start states of the NFA's N_i for pattern P_i .



Example 3.26

Note that these three patterns present some conflicts of the type

In particular, string ***abb*** matches both the second and third patterns,

lexeme for pattern P_2 , since that pattern is listed first in the above Lexer program.

Then, input strings such as ***aabbb…***

have many prefixes that match the third pattern. The Lex rule is to take the longest, so we continue reading b's , until another a is met.

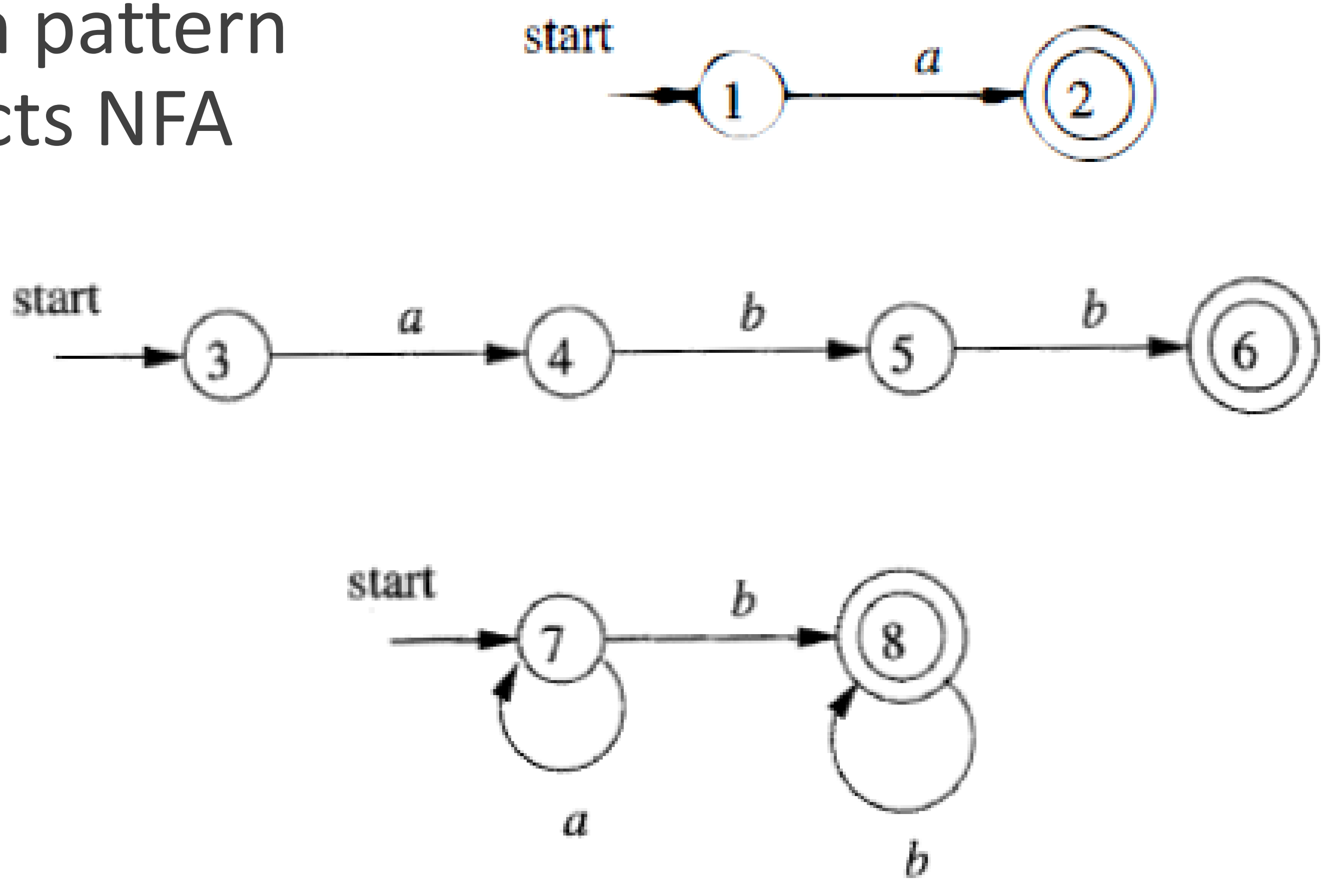
a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
$\mathbf{a^*b^+}$	{ action A_3 for pattern p_3 }

Conflict Resolving

1. Find the longest matching token
2. Between two tokens with the same length use the one declared first

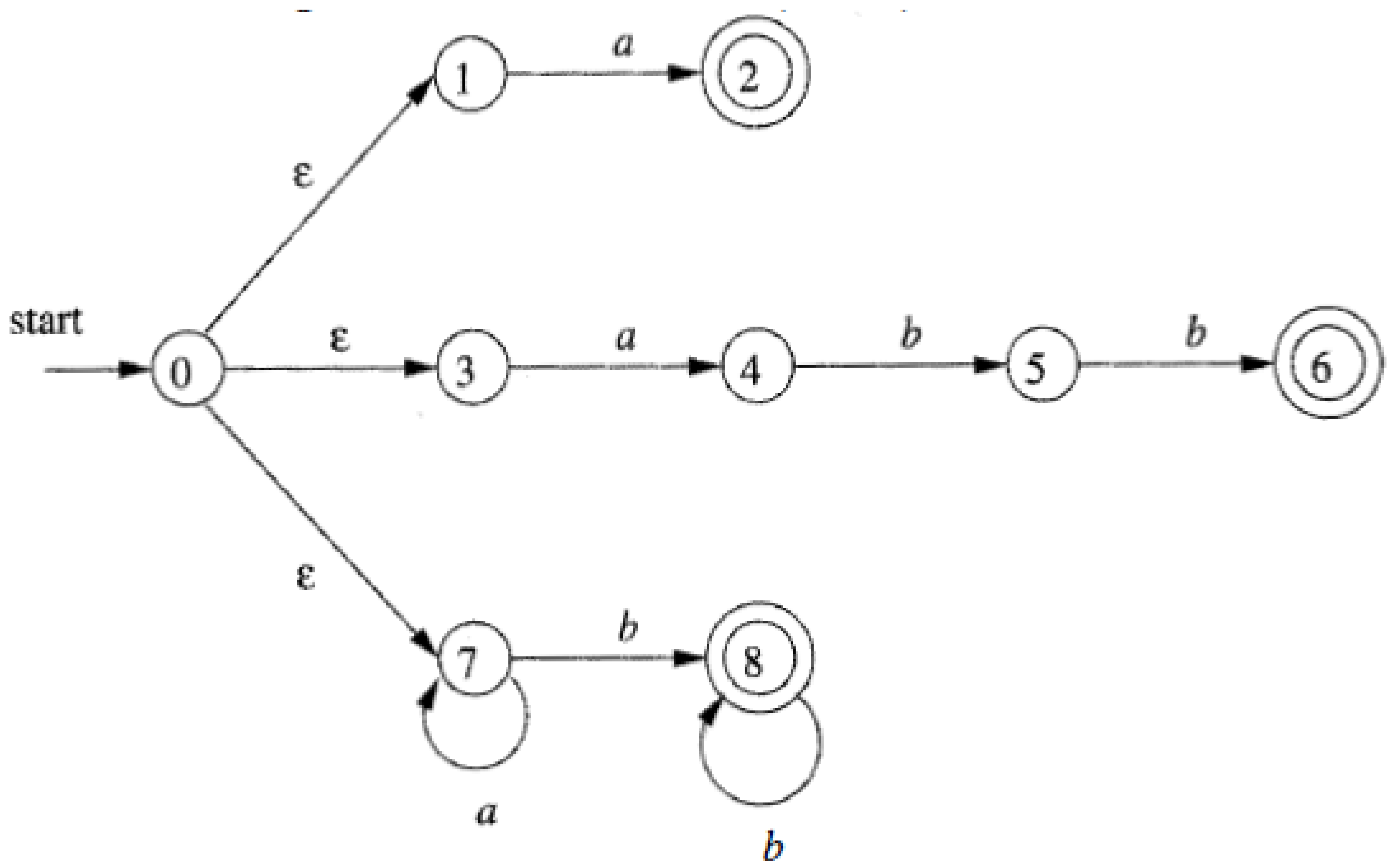
Example 3.26

For each pattern constructs NFA



Example 3.26

Combine all NFA's



Example 3.26

1. Read input beginning and referred to it as ***lexemeBegin***.
2. As it moves the pointer called ***forward*** ahead in the input ,
3. At each point calculates the ***set of states***.
4. The NFA simulation reaches a point on the input where there are ***no next states***.
5. ***look backwards*** in the ***sequence of sets of states***, until find a set that includes one or more ***accepting states***.
6. 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.
7. Move the ***forward*** pointer back to the ***end*** of the ***lexeme***, and start over.

Example 3.26

X = aaba

Starting with t-closure of the initial state 0, which is {0, 1, 3, 7 }.

{0,1,3,7}, a -> {2,4,7} –>Ɛ*-> {2,4,7}

{2,4,7}, a -> {7} –>Ɛ*-> {7}

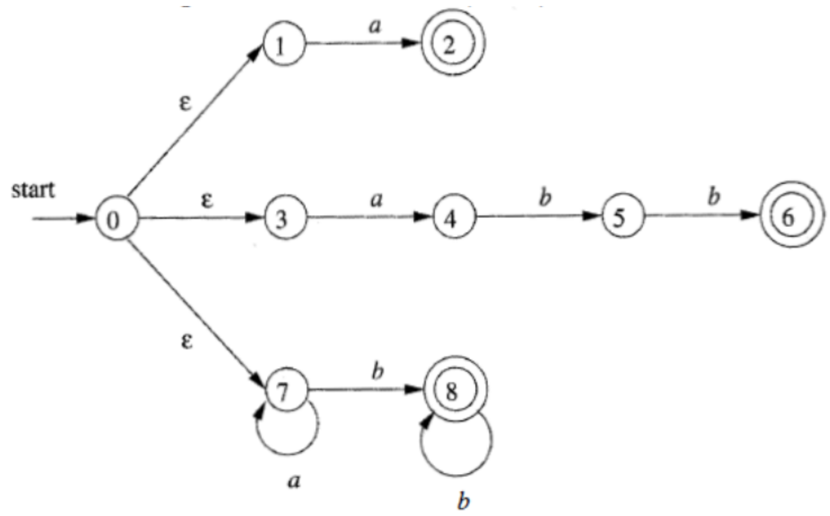
{7}, b -> {8} –>Ɛ*-> {8}

{8}, a -> ø

{0,1,3,7}, {2,4,7}, {7}, {8}, ø

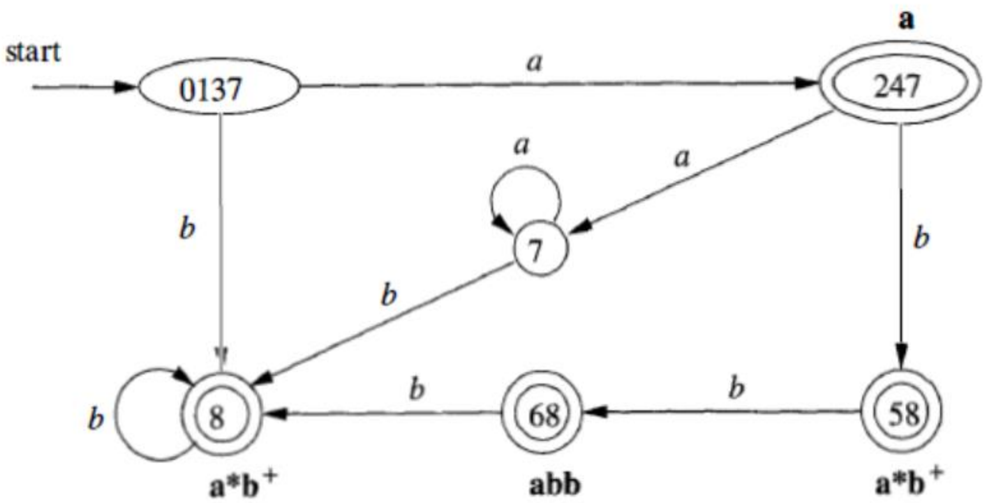
Which one (2,a), or (8, aab)?

Longest (8, aab) and then start from “a” again.



Example 3.26

- NFA to DFA
- And omit dead states
- 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 ***abb*** is listed first, that is the pattern associated with state {6, 8 }.



Example 3.26

X = abba

Start from state 0137

The sequence of states entered is
0137,a -> 247

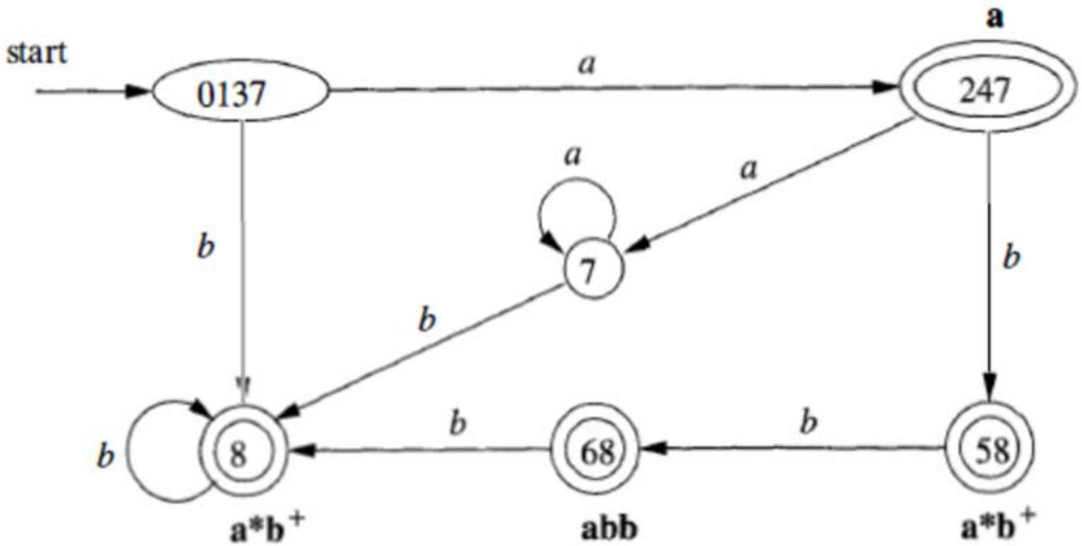
247, b -> 58

58, b -> 68

68 , a -> ø

0137, 247, 58, 68

pattern P_2 = abb.



Efficiency of Algorithms

- The cost of converting a regular expression ***"r"*** to an NFA is ***O(|r|)***, where ***|r|*** stands for the size of ***"r"***.
- With at most ***|r| states*** and at most ***2|r| edges.***
- NFA to DFA : For every ***DFA state*** constructed, we must construct at most ***|r|*** new states, and each one takes at most ***O(|r| + 2 |r|)*** time.
- The time to construct a DFA of ***"s"*** states is thus ***O((|r|^{2}s).***
- Common case where ***"s"*** is about ***|r|.***
- Worst case where ***"s"*** is about ***2$^{|r|}$.***

AUTOMATON	INITIAL	PER STRING
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Minimizing the Number of States of a DFA

1. Start with an initial partition Π with two groups, F and $S - F$, the accepting and nonaccepting states of D.

2. Apply the procedure of Fig. 3.64 to construct a new partition Π_{new}.

```
initially, let Π_new = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Π_new by the set of all subgroups formed;
}
```

Minimizing the Number of States of a DFA

3. If $\Pi_{\text{new}} = \Pi$, let $\Pi_{\text{final}} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π.

4. Choose one state in each group of Π_{final} as the *representative* for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows:

Example

Two classes : {0,1,2,3}, {4}

(0,a->1), (1,a->1), (2,a->1), (3,a->1)

(0,b->2), (1,b->3), (2,b->2), (3,b->4)

New classes {0,1,2}, {3}, {4}

(0,a->1), (1,a->1), (2,a->1)

(0,b->2), (1,b->3), (2,b->2)

New classes {0, 2}, {1}, {3}, {4}

(0,a->1), (2,a->1) no change

(0,b->2), (2,b->2) no change

Last classes {0, 2}, {1}, {3}, {4}