Chapter 1: Regular Expressions & Lexical Analysis

Yepang Liu

liuyp1@sustech.edu.cn

The chapter numbering in lecture notes does not follow that in the textbook.

Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- Finite Automata (for recognizing patterns)

  - NFA & DFA
  - NFA → DFA
  - Regexp → NFA
  - Combining NFAs

Finite Automata (有穷自动机)

- Finite automata are the simplest machines to recognize patterns
- They take a string as input and output "yes" (pattern is matched) or "no" (pattern is unmatched).
  - **Nondeterministic finite automata (NFA, 非确定有穷自动机):** A symbol can label several edges out of the same state (allowing multiple target states), and the empty string ϵ is a possible label.
  - **Deterministic finite automata (DFA, 确定有穷自动机):** For each state and for each symbol in the input alphabet, there is exactly one edge with that symbol leaving that state.
- NFA and DFA recognize the same languages, **regular languages**, which regexps can describe.

Nondeterministic Finite Automata

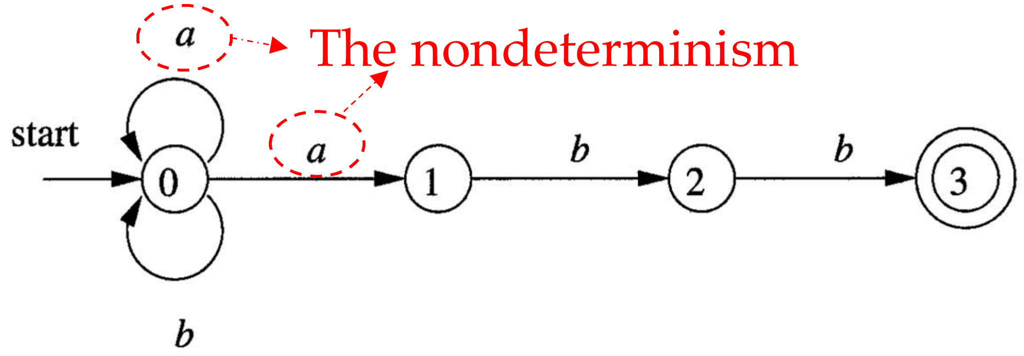
- An **NFA** is a 5-tuple, consisting of:

  1. A finite set of states S
  2. A set of input symbols Σ, the *input alphabet*. We assume that the empty string ϵ is never a member of Σ
  3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*
  4. A *start state* (or initial state) s_0 from S
  5. A set of *accepting states* (or *final states*) F, a subset of S

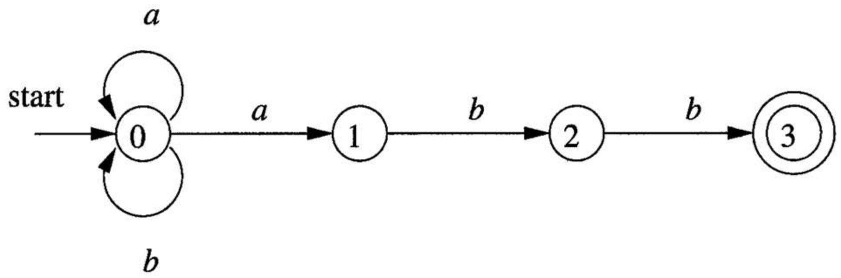
NFA Example

- S = {0, 1, 2, 3}
- Σ = {a, b}
- Start state: 0
- Accepting states: {3}
- Transition function
  - (0, a) → {0, 1}   (0, b) → {0}
  - (1, b) → {2}   (2, b) → {3}

The NFA can be represented as a Transition Graph:



Transition Table

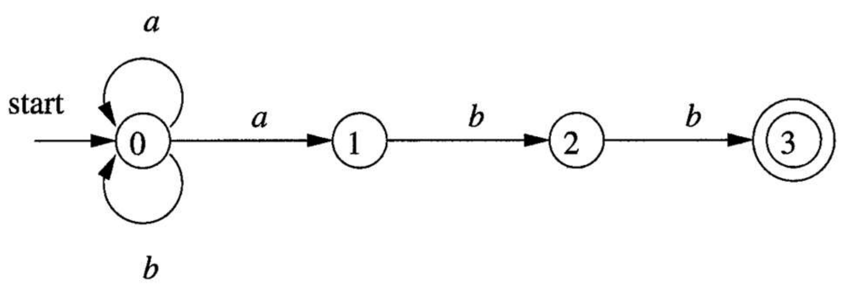


- Another representation of an NFA
  - Rows correspond to states
  - Columns correspond to the input symbols or ϵ
  - The table entry for a state-input pair lists the set of next states
  - $\emptyset$: the transition function has no information about the state-input pair (the move is not allowed, there is an error during recognition)

STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Acceptance of Input Strings

- An NFA accepts an input string x if and only if
  - There is a path in the transition graph from the start state to one accepting state, such that the symbols along the path form x (ϵ labels are ignored).

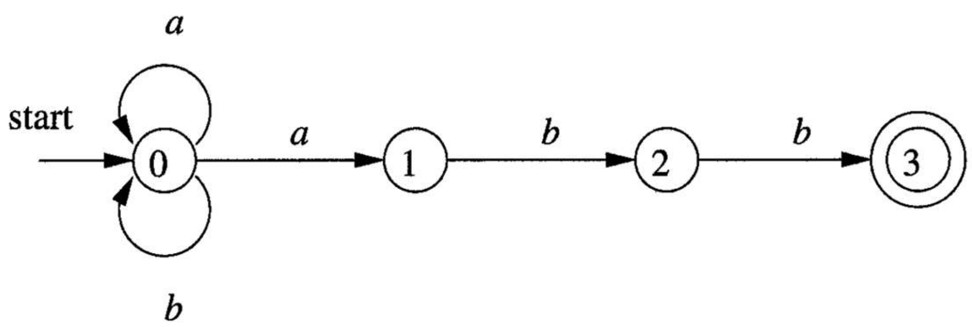


accepts the string *aabb*

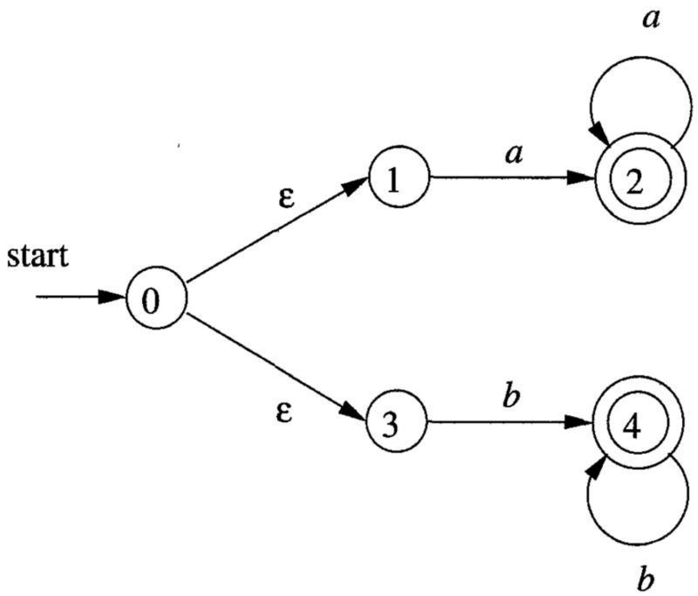
- The language defined or accepted by an NFA
  - The set of strings labelling some path from the start state to an accepting state

NFA and Regular Languages

What language can be accepted by each of the following NFAs?



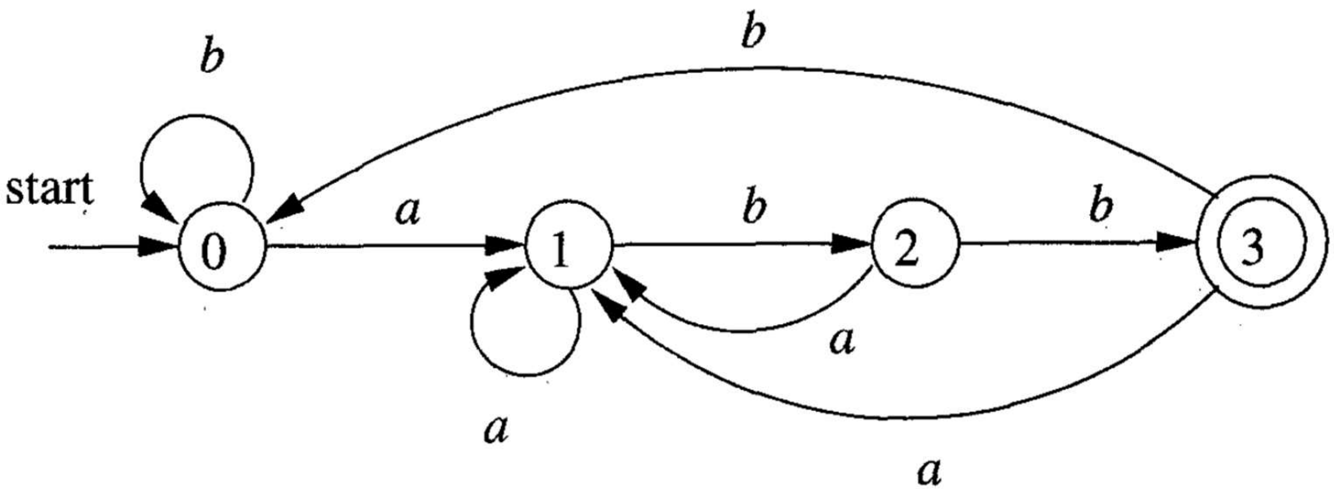
$L((a|b)^*abb)$



$L(aa^*|bb^*)$

Deterministic Finite Automata (DFA)

- A **DFA** is a special NFA where:
  - There are no moves on input ϵ
  - For each state s and input symbol a, there is exactly one edge out of s labeled a (i.e., exactly one target state)



Simulating a DFA

- **Input:**
  - String x terminated by an end-of-file character **eof**.
  - DFA D with *start state* s_0, *accepting states* F, and transition function *move*
- **Output:** Answer "yes" if D accepts x; "no" otherwise

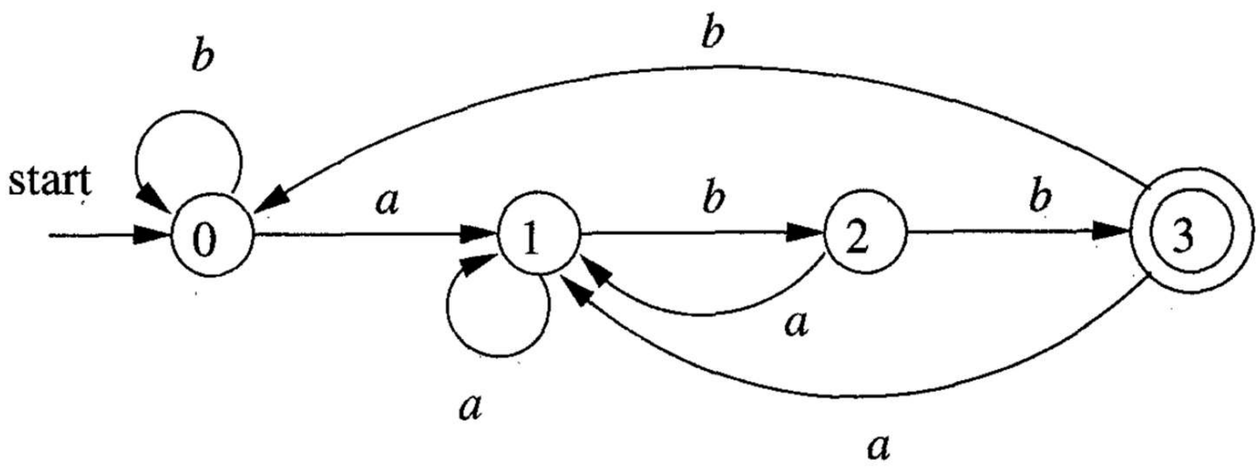
```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

We can see from the algorithm:

- DFA can efficiently accept/reject strings (i.e., recognize patterns)

DFA Example

- Given the input string *ababb*, the DFA below enters the sequence of states 0, 1, 2, 1, 2, 3 and returns "yes"



What's the language defined by this DFA?

Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- Finite Automata (for recognizing patterns)
  - NFA & DFA
  - NFA → DFA
  - Regexp → NFA
  - Combining NFAs

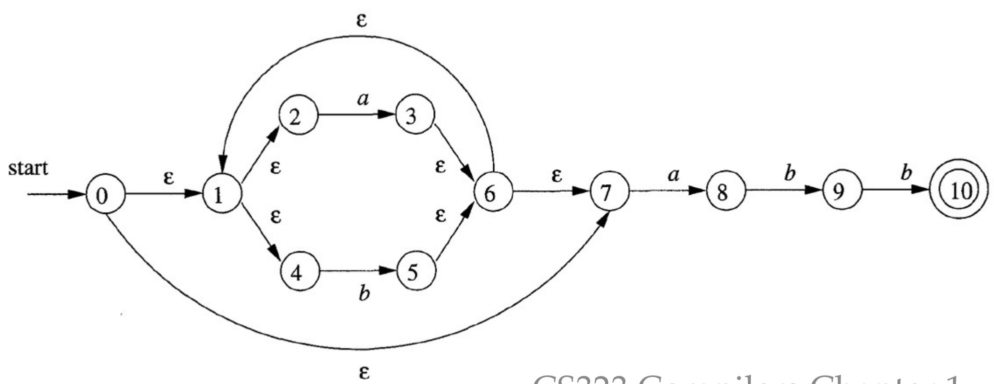
From Regular Expressions to Automata

- Regexps concisely & precisely describe the patterns of tokens
- DFA can efficiently recognize patterns (comparatively, the simulation of NFA is less straightforward*)
- When implementing lexical analyzers, regexps are often converted to DFA:
  - Regexp → NFA → DFA
  - Algorithms: Thompson's construction + subset construction

* There may be multiple transitions at a state when seeing a symbol

Conversion of an NFA to a DFA

- The **subset construction** algorithm (子集构造法)
  - **Insight:** Each state of the constructed DFA corresponds to a set of NFA states
    - Why? Because after reading the input $a_1a_2\ldots a_n$, the DFA reaches one state while the NFA may reach multiple states
  - **Basic idea:** The algorithm simulates "in parallel" all possible moves an NFA can make on a given input string to map a set of NFA states to a DFA state.

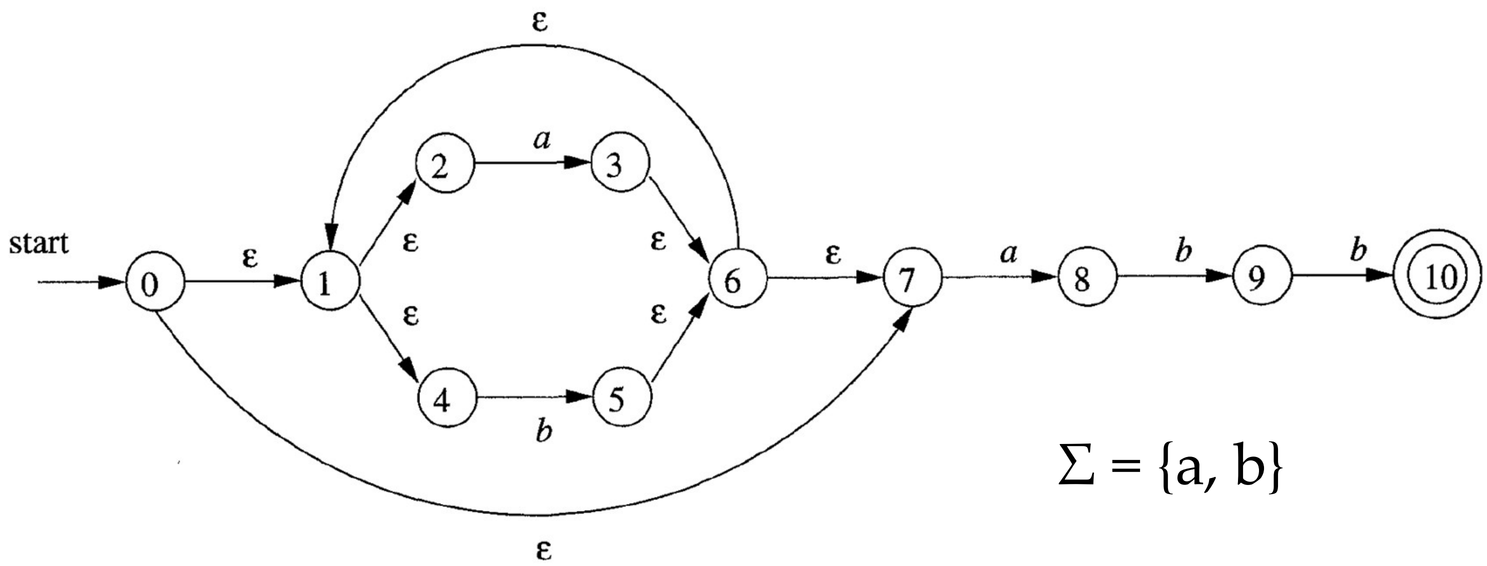


After reading "a", the NFA may reach any of these states:

3, 6, 1, 7, 2, 4, 8

Example for Algorithm Illustration

- The NFA below accepts the string *babb*
  - There exists a path from the start state 0 to the accepting state 10, on which the labels on the edges form the string *babb*



$\Sigma = \{a, b\}$

Subset Construction Technique

- Operations used in the algorithm:
  - ***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
    - That is, $\bigcup_{s\ in\ T} \epsilon\text{–}closure(s)$
  - ***move*(*T, a*):** Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T* (i.e., the target states of those states in *T* when seeing *a*)

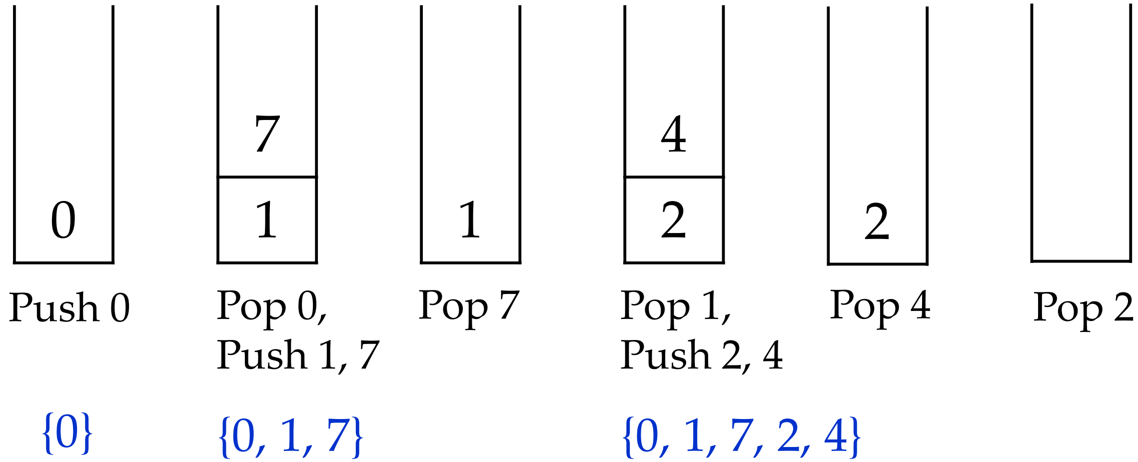
Subset Construction Technique

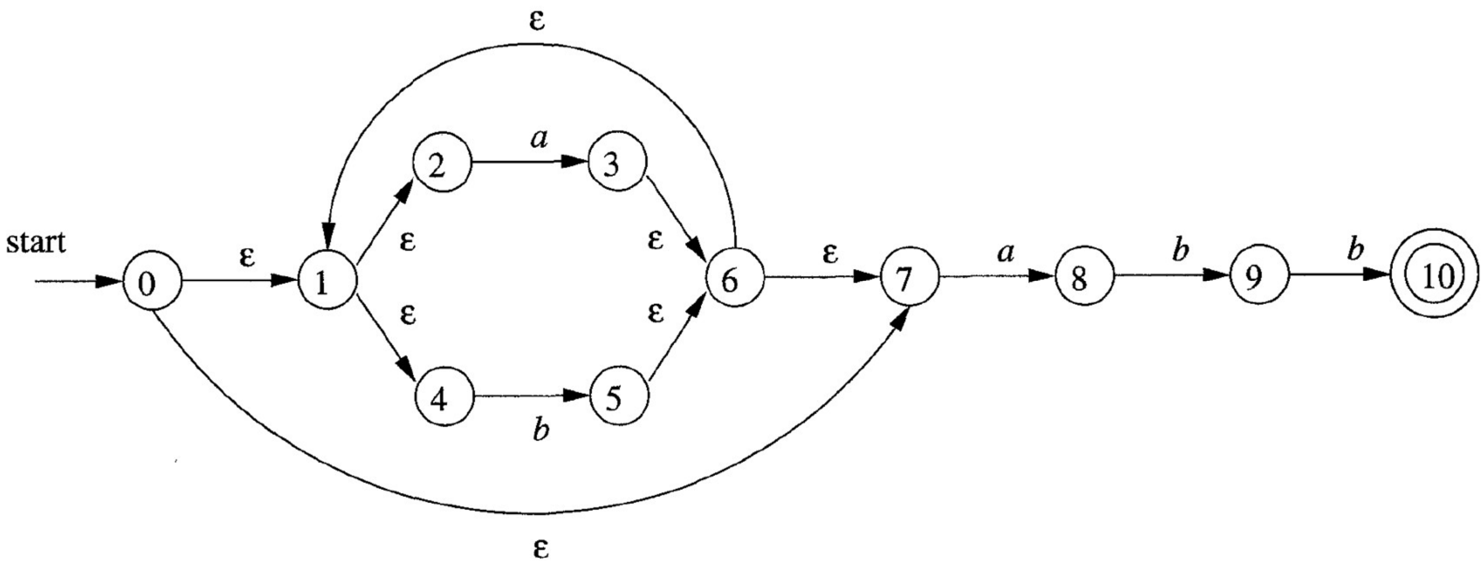
- **Computing *ϵ-closure*(T)**
  - It is a graph traversal process (only consider ϵ edges)
  - Computing *ϵ-closure*(s) is the same (when T has only one state)

```
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;
                }
}
```

Illustrative Example

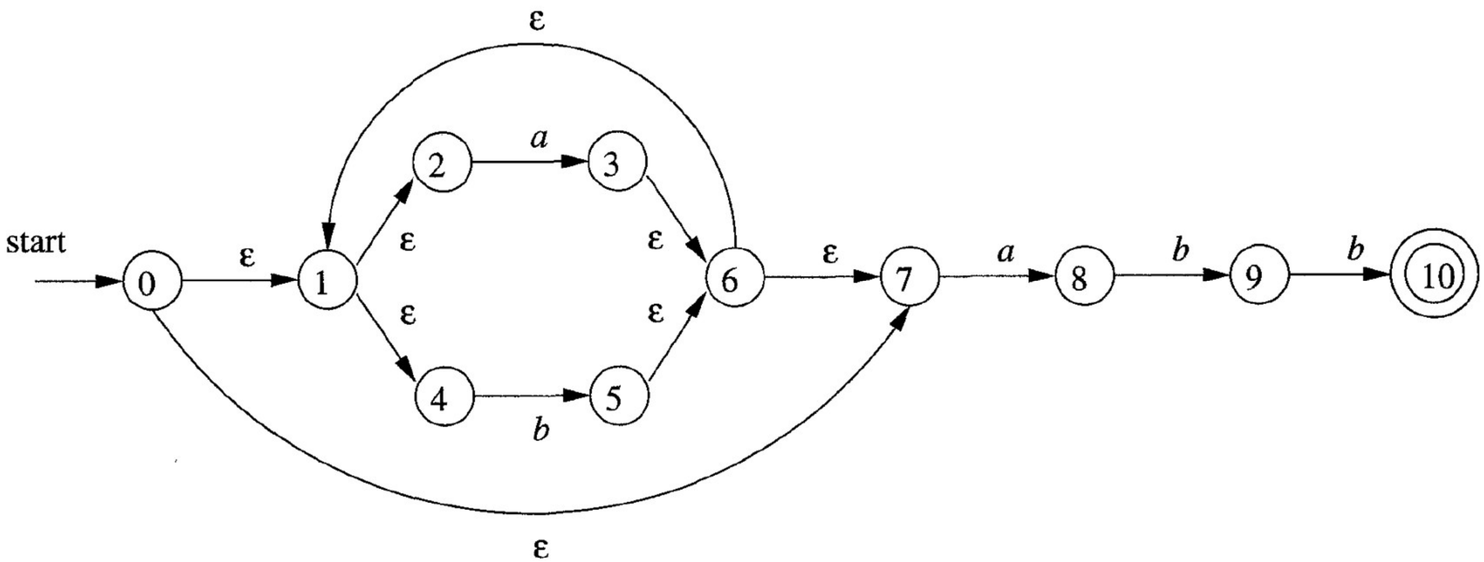
- ϵ-*closure*(0) = ?





Exercise (Please do this after class)

- ϵ-*closure*({3, 8}) = ?



Subset Construction Technique Cont.

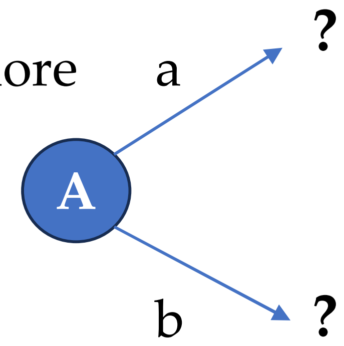
- The construction of the DFA *D*'s states, *Dstates*, and the transition function *Dtran* is also a search process
  - Initially, the only state in *Dstates* is ϵ-*closure*(s_0) and it is unmarked
    - Unmarked state means that its next states have not been explored

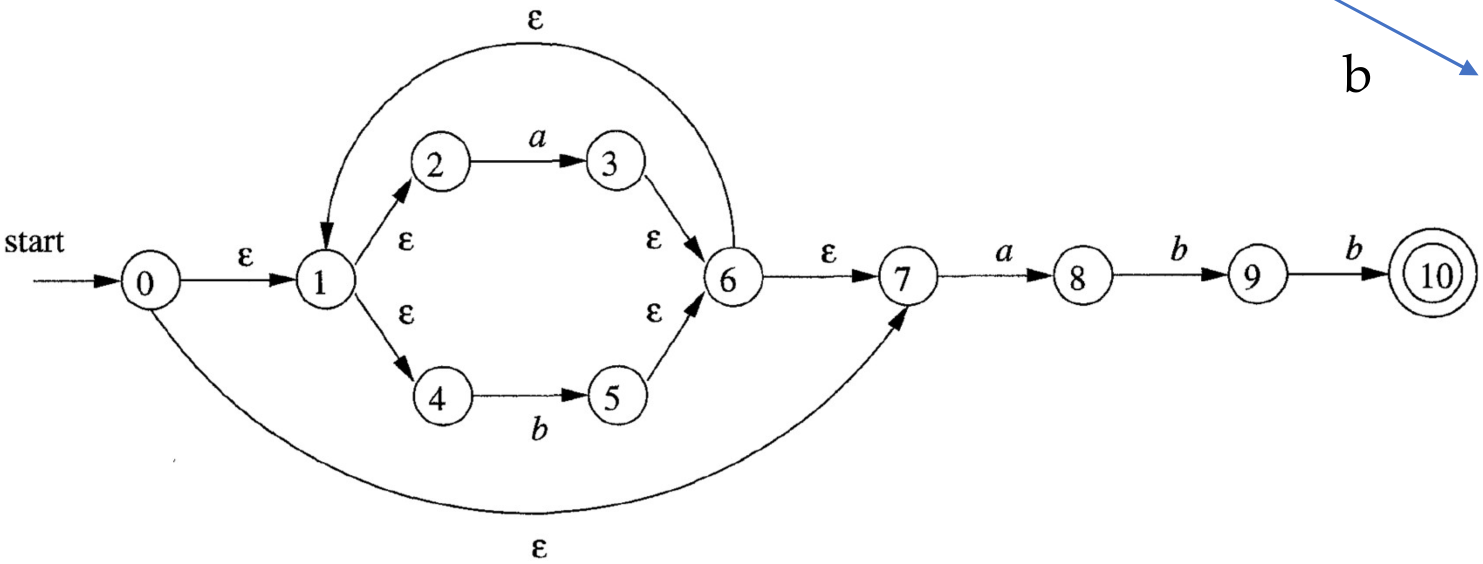
```
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {  // find the next states of T
                U = ϵ-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Illustrative Example

- Initially, Dstates only has one unmarked state:
  - *ϵ-closure*(0) = {0, 1, 2, 4, 7} -- A
- Dtran is empty

The next step is to explore





Illustrative Example

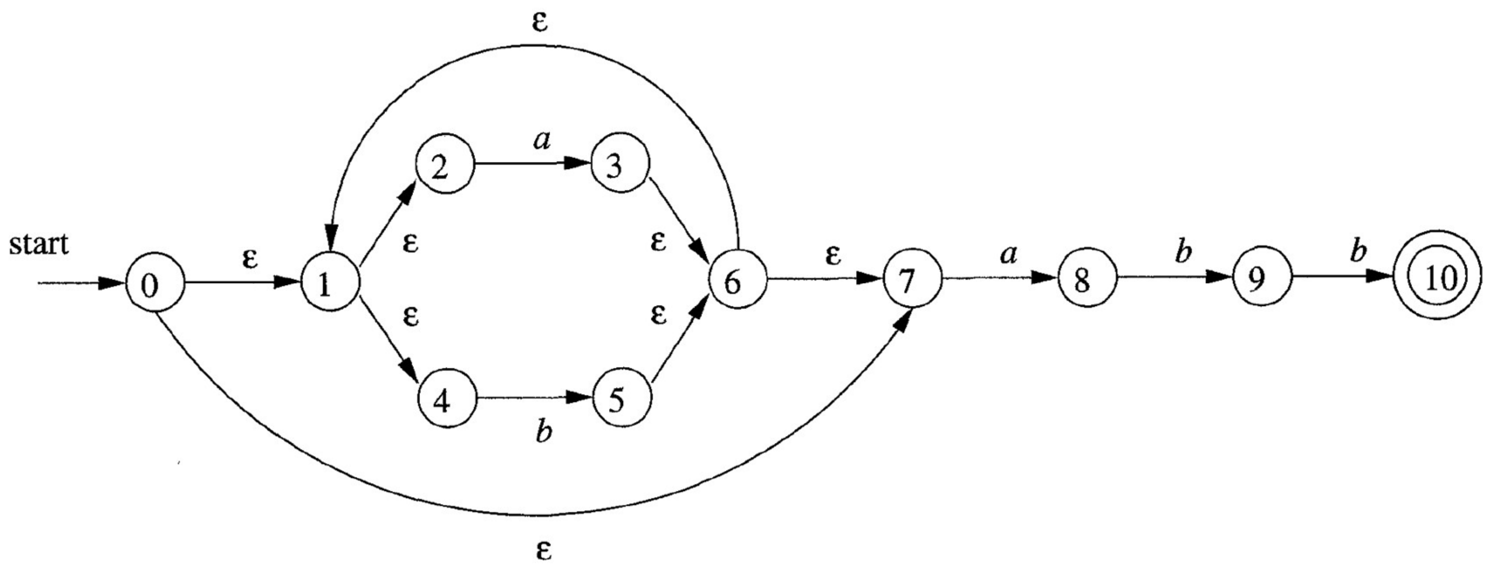
{0, 1, 2, 4, 7} -- A

ϵ-*closure*(*move*[A, a])

= ϵ-*closure*({3, 8})

= {1, 2, 3, 4, 6, 7, 8}

- We get an unseen state {1, 2, 3, 4, 6, 7, 8} -- B
- Update Dstates: {A, B}
- Update Dtran: {[A, a] ➔ B}



Illustrative Example

{0, 1, 2, 4, 7} -- A

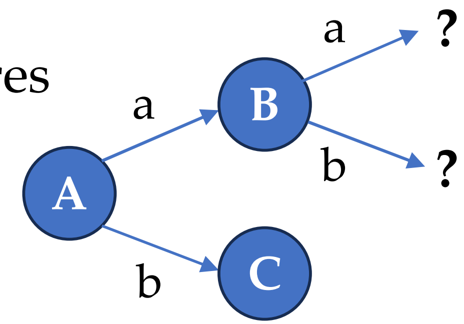
ϵ-*closure*(*move*[A, b])

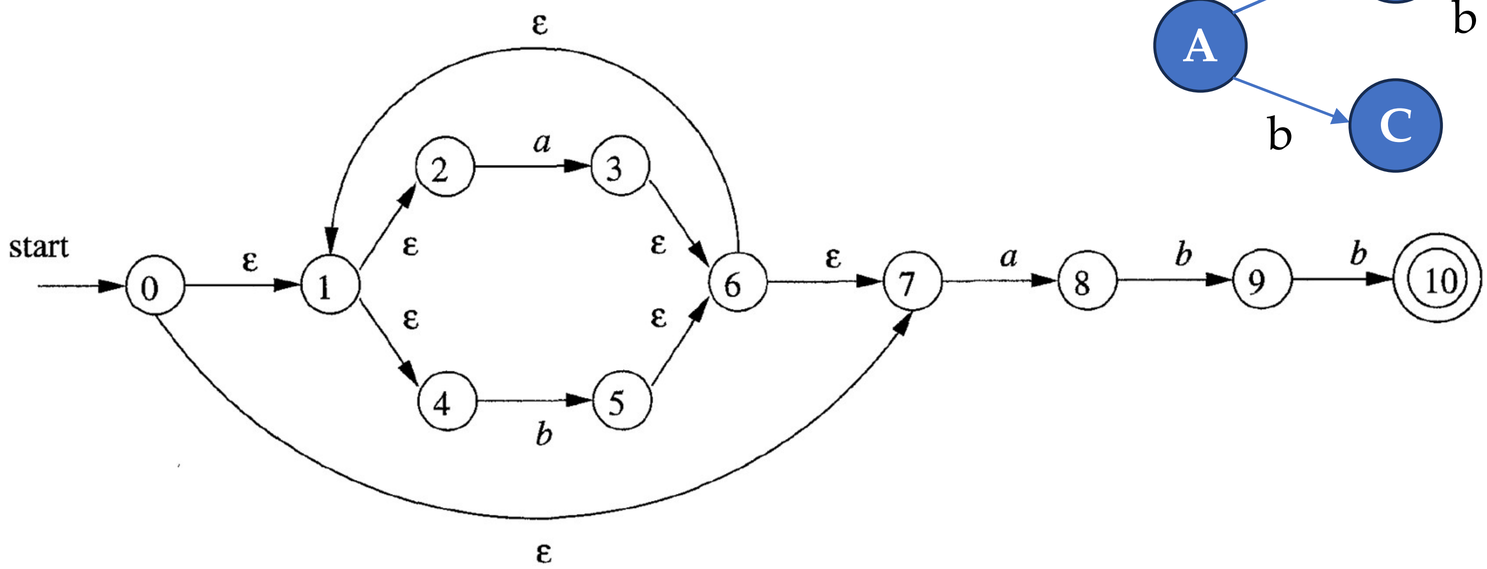
= ϵ-*closure*({5})

= {1, 2, 4, 5, 6, 7}

- We get an unseen state {1, 2, 4, 5, 6, 7} -- C
- Update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C}

After A is fully explored (marked), the next step explores





Illustrative Example

{1, 2, 3, 4, 6, 7, 8} -- B

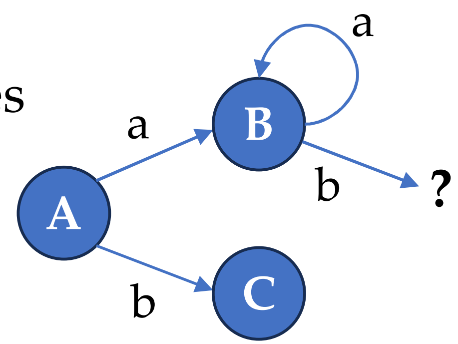
ϵ-closure(*move*[B, a])

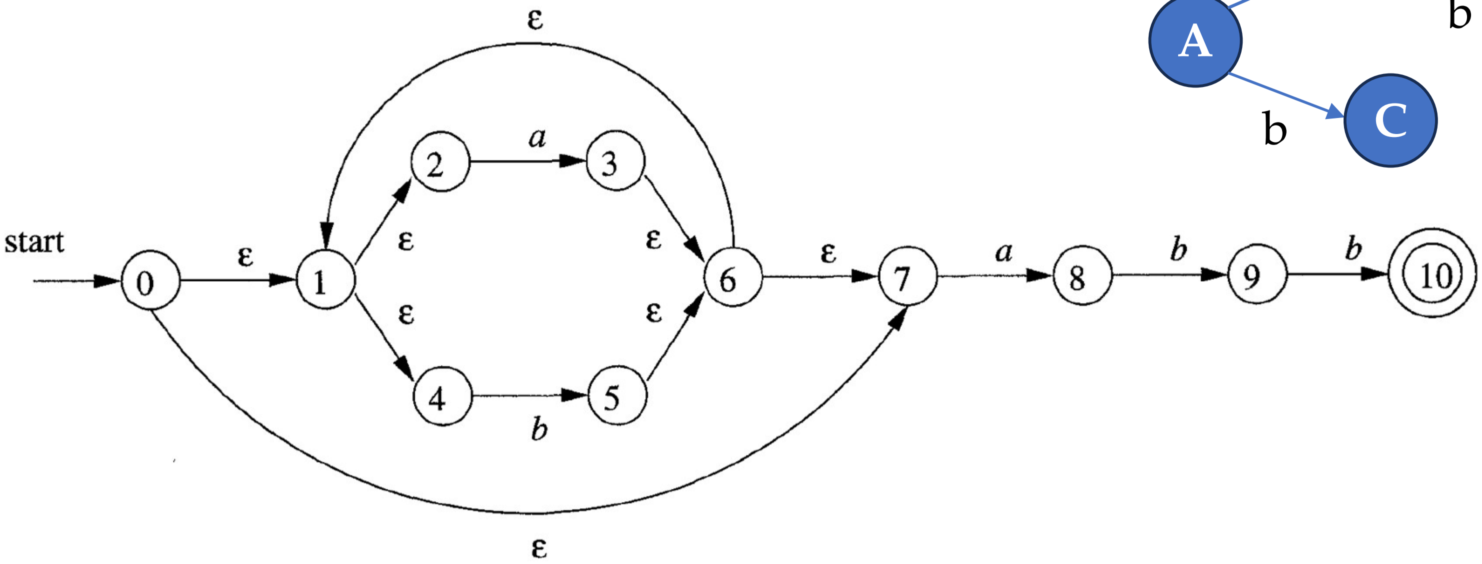
= *ϵ-closure*({3, 8})

= {1, 2, 3, 4, 6, 7, 8}

- The state {1, 2, 3, 4, 6, 7, 8} already exists (B)
- No need to update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C, [B, a] ➔ B}

The next step explores





Illustrative Example

- Eventually, we will get the following DFA:
  - Start state: A;   Accepting states: {E}

NFA State	DFA State	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 4, 5, 6, 7, 10\}$	E	B	C

This DFA can be further minimized: A and C have the same moves on all symbols and can be merged.

Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- Finite Automata (for recognizing patterns)
  - NFA & DFA
  - NFA → DFA
  - Regexp → NFA
  - Combining NFAs

Regular Expression to NFA

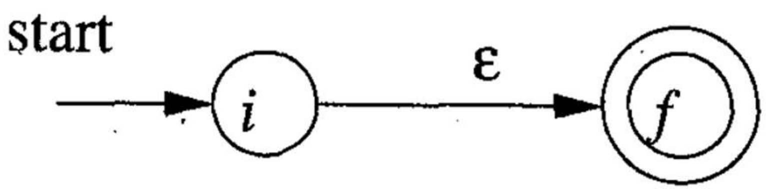
Thompson's construction algorithm (Thompson构造法)

- The algorithm works recursively by splitting a regular expression into subexpressions, from which the NFA will be constructed using the following rules:
  - **Two basis rules (基本规则):** handle basic expressions without any operators
  - **Three inductive rules (归纳规则):** construct larger NFAs from the smaller NFAs for subexpressions

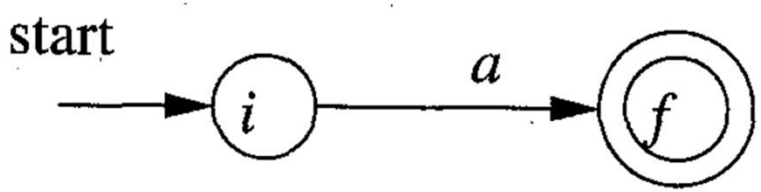
Thompson's Construction Algorithm

Two basis rules:

1. The empty expression ϵ is converted to



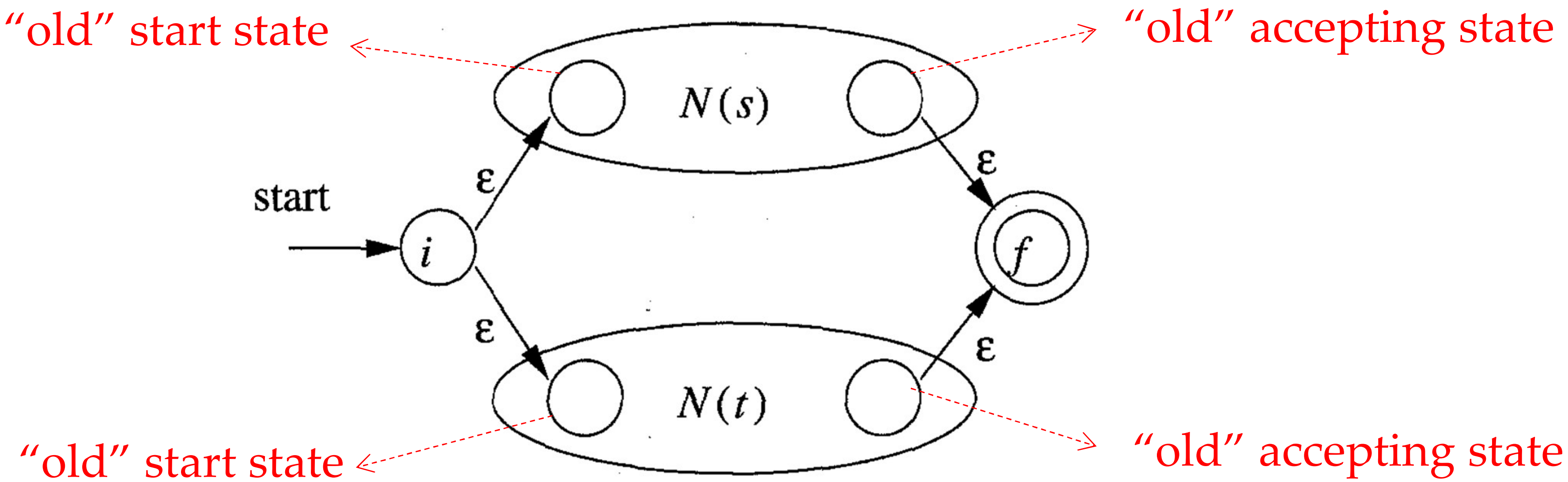
2. Any subexpression a (a single symbol in input alphabet) is converted to



Thompson's Construction Algorithm

Inductive rule #1: the union case

- ***s* | *t* :** $N(s)$ and $N(t)$ are NFAs for subexpressions s and t

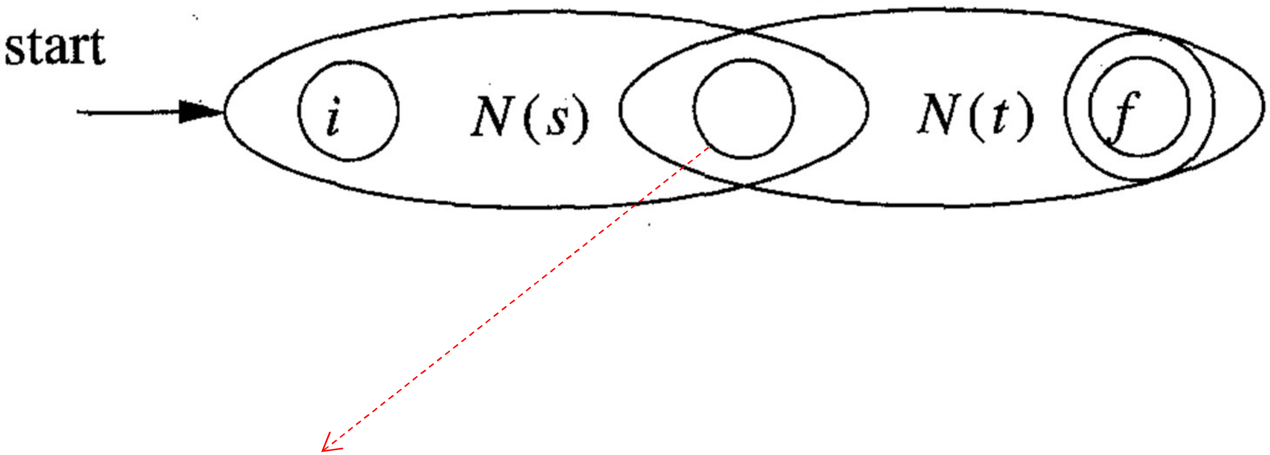


By construction, the NFAs have only one start state and one accepting state

Thompson's Construction Algorithm

Inductive rule #2: the concatenation case

- ***st*** **:** $N(s)$ and $N(t)$ are NFAs for subexpressions s and t

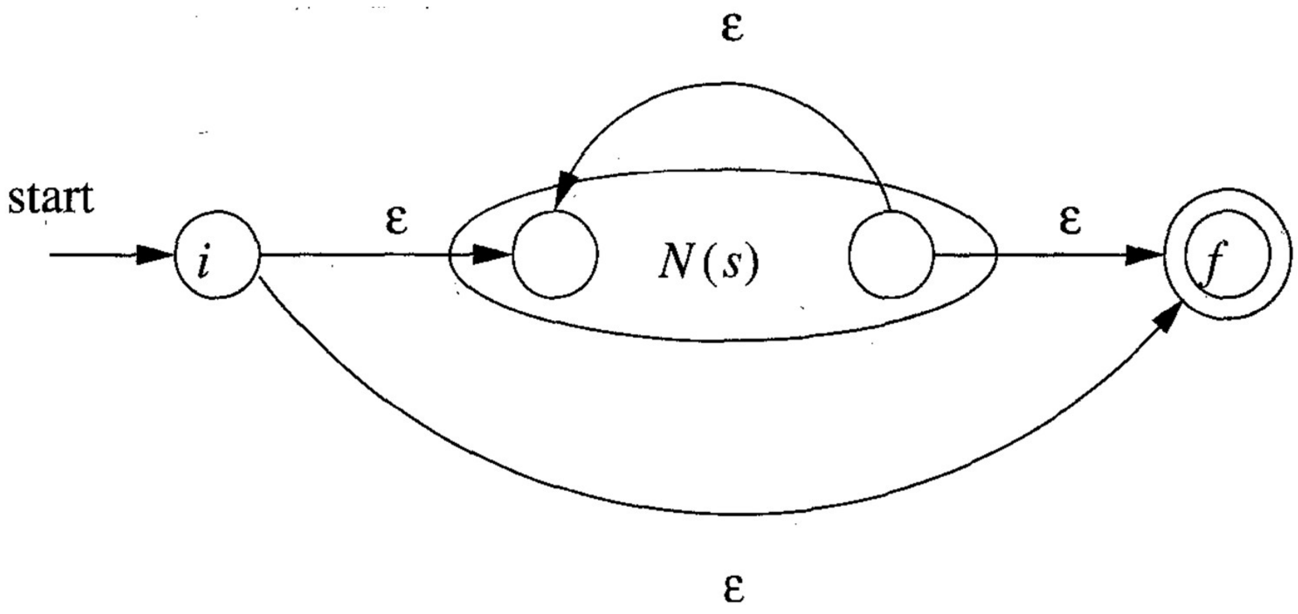


Merging the accepting state of $N(s)$ and the start state of $N(t)$

Thompson's Construction Algorithm

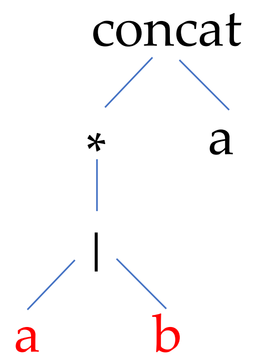
Inductive rule #3: the Kleene closure case

- s^* **:** $N(s)$ is the NFA for subexpression s

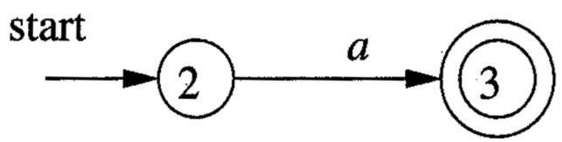


Example

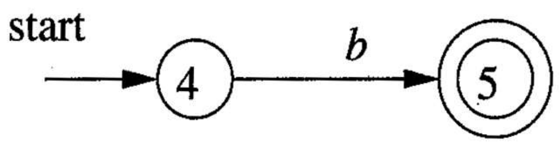
Use Thompson's algorithm to construct an NFA for the regexp r = **(a | b)*a**



1. NFA for the first **a** (apply basis rule #1)

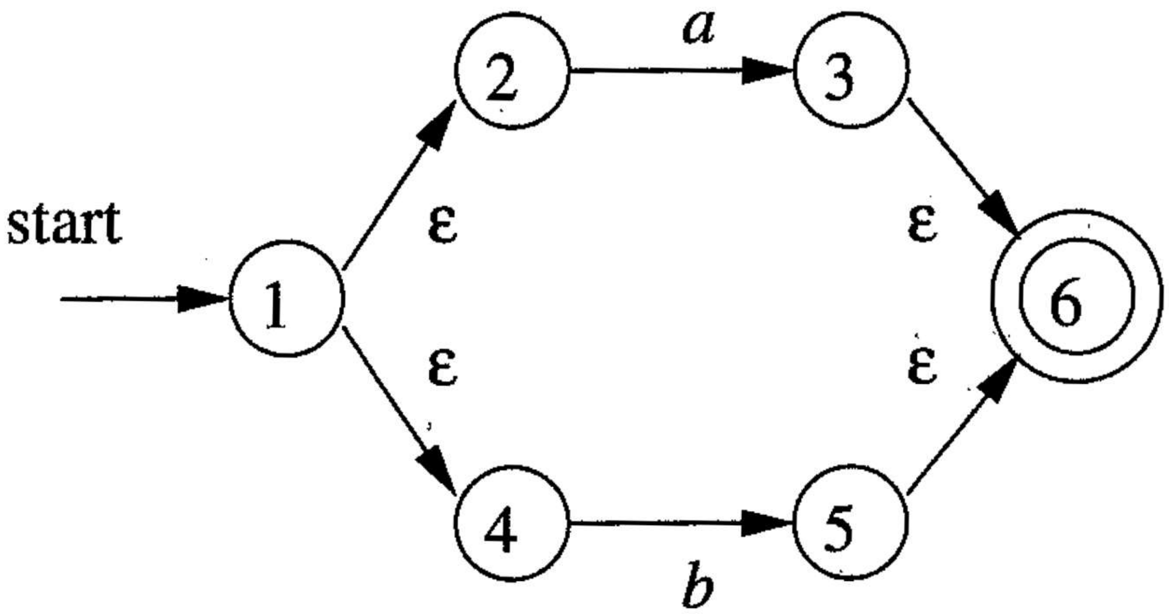


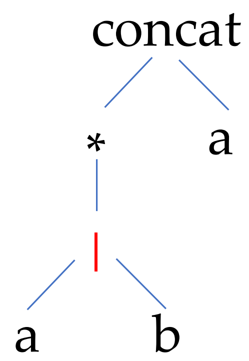
2. NFA for the first **b** (apply basis rule #1)



Example $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$

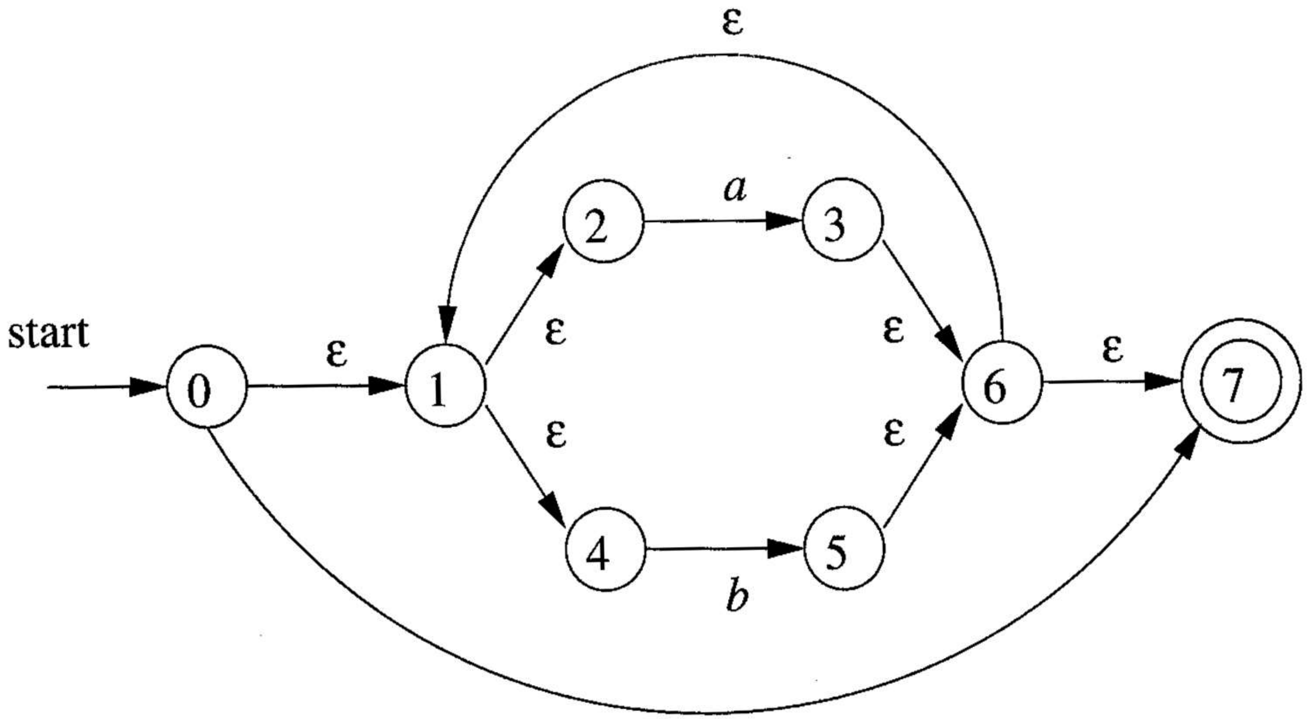
3. NFA for **(a | b)** (apply inductive rule #1)

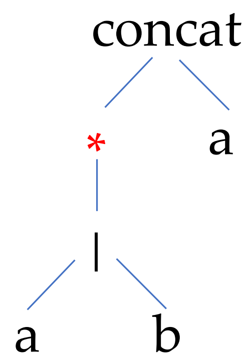




Example $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$

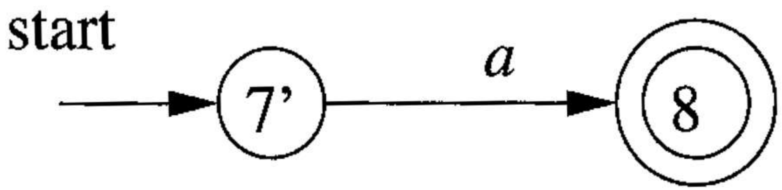
4. NFA for $(\mathbf{a} \mid \mathbf{b})^*$ (apply inductive rule #3)

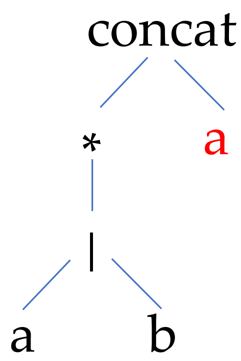




Example $r = (\mathbf{a} \mid \mathbf{b})^{*}\mathbf{a}$

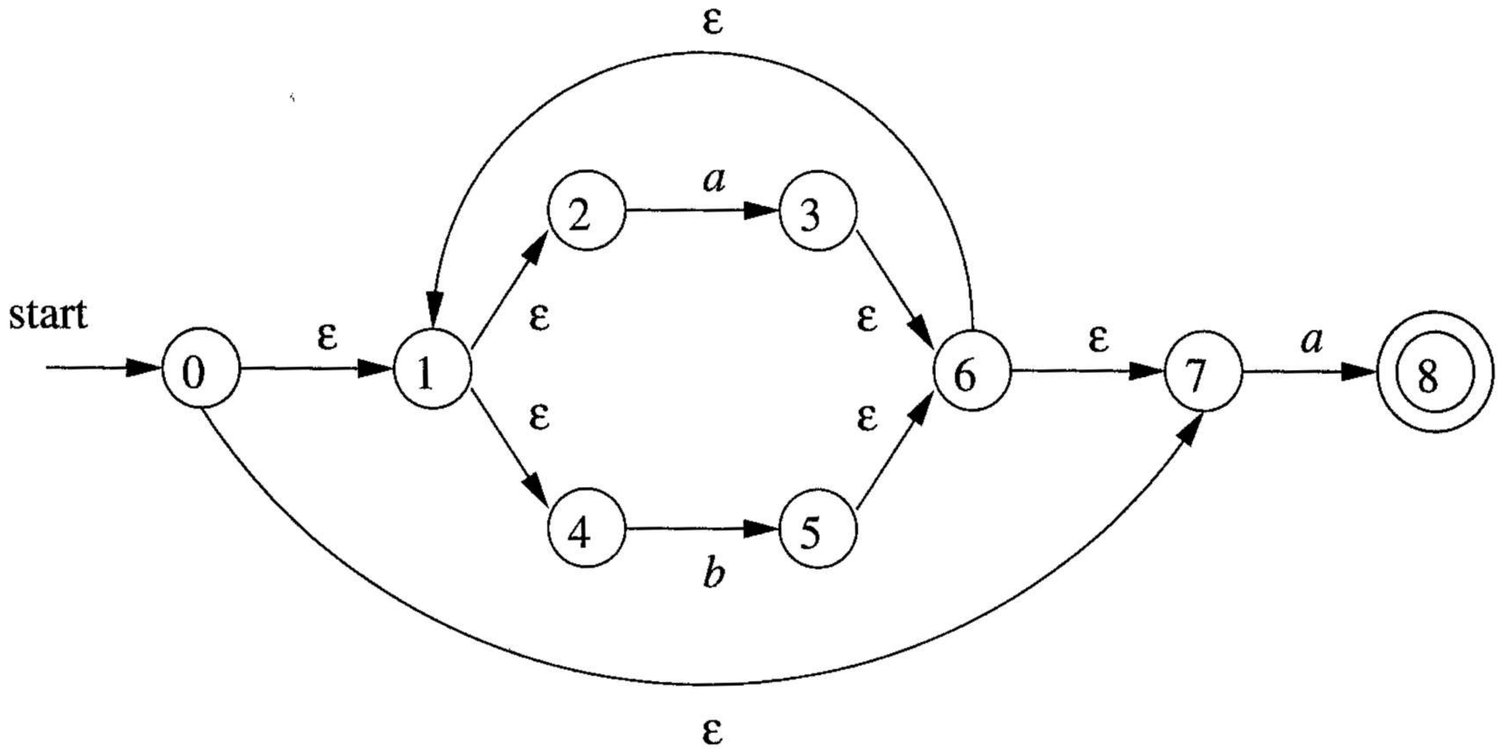
5. NFA for the second **a** (apply basis rule #1)

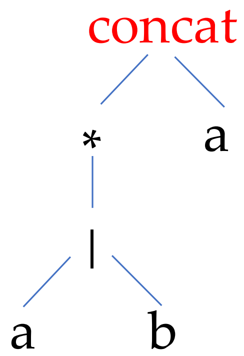




Example $r = (\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$

6. NFA for $(\mathbf{a} \mid \mathbf{b})^* \mathbf{a}$ (apply inductive rule #2)



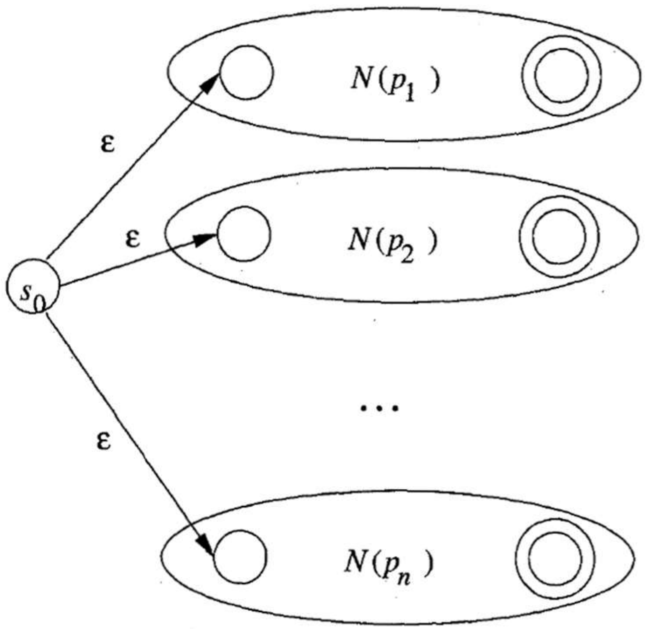


Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- Finite Automata (for recognizing patterns)
    - NFA & DFA
    - NFA → DFA
    - Regexp → NFA
    - Combining NFAs

Combining NFAs

- **Why?** In the lexical analyzer, we need a single automaton to recognize lexemes matching any pattern
- **How?** Introduce a new start state with ϵ-transitions to each of the start states of the NFAs for pattern p_i



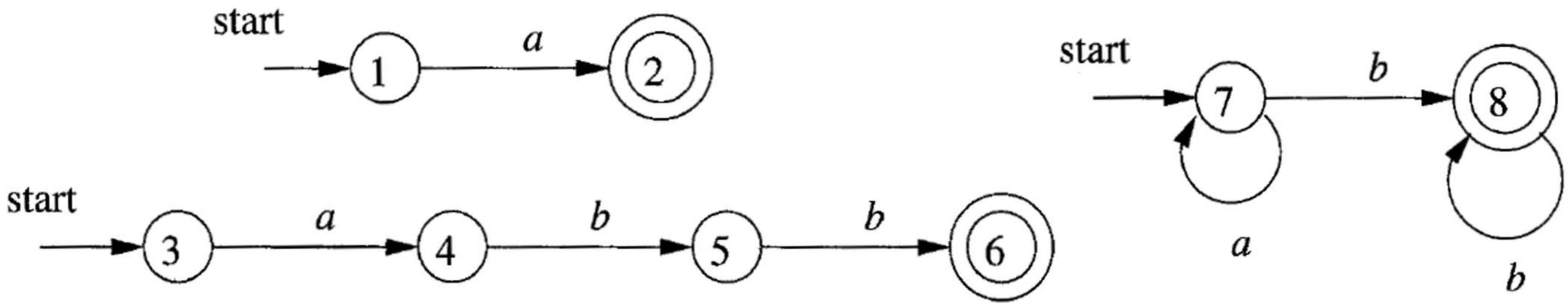
- The language that can be accepted by the big NFA is the union of the languages that can be accepted by the small NFAs
- Different accepting states correspond to different patterns

DFAs for Lexical Analyzers

- Convert the NFA for all the patterns into an equivalent DFA, using the subset construction algorithm
- An accepting state of the DFA corresponds to a subset of the NFA states, in which at least one is an accepting NFA state
  - If there are more than one accepting NFA state, this means that conflicts arise (the prefix of the input string matches multiple patterns)
  - Upon conflicts, find the first pattern whose accepting state is in the set and make that pattern the output of the DFA state

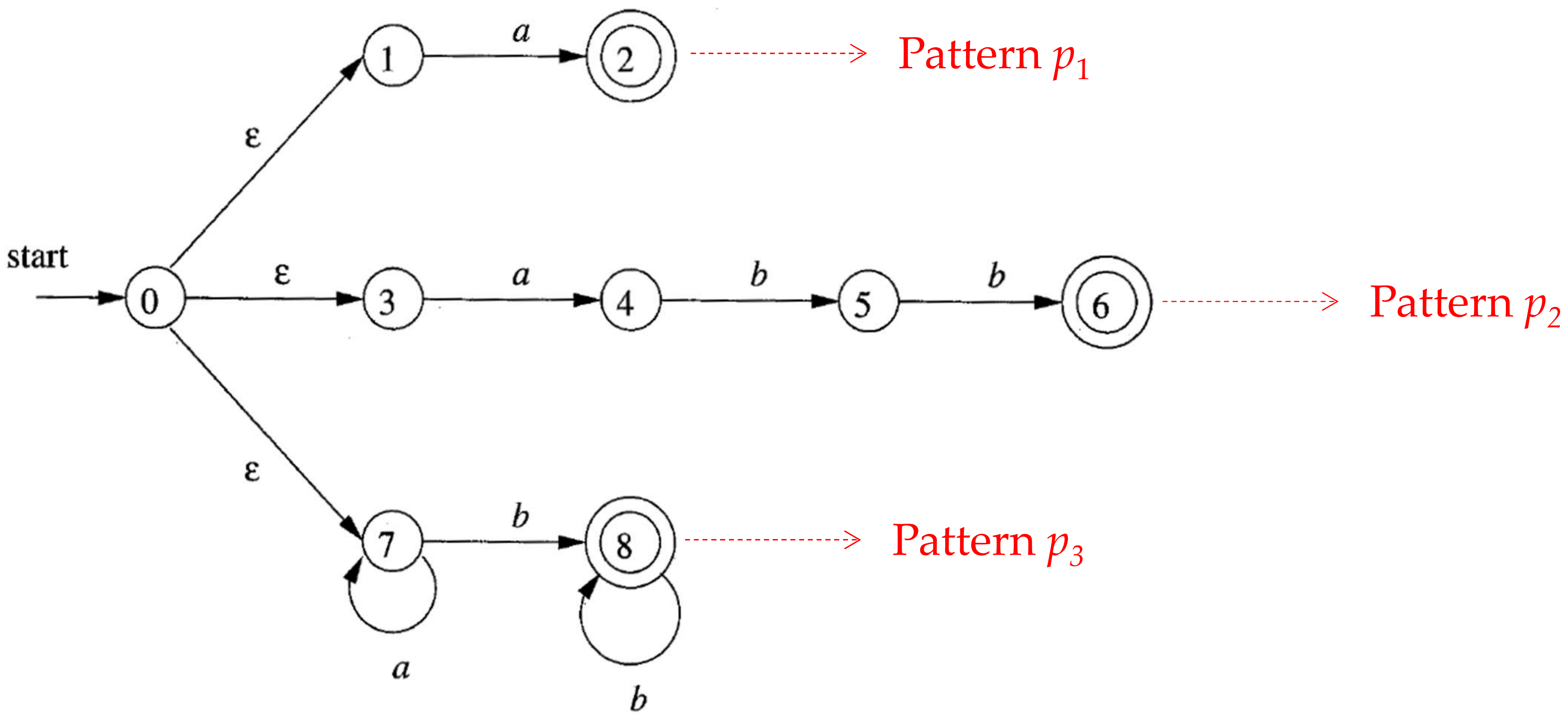
Example

- Suppose we have three patterns: p_1, p_2, and p_3
  - **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}
- We first construct an NFA for each pattern



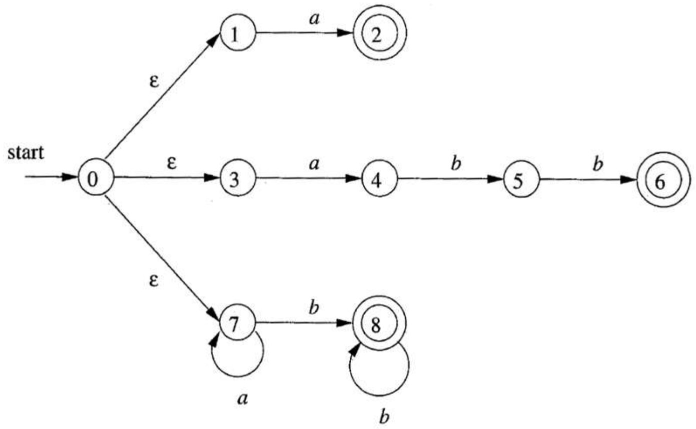
Example

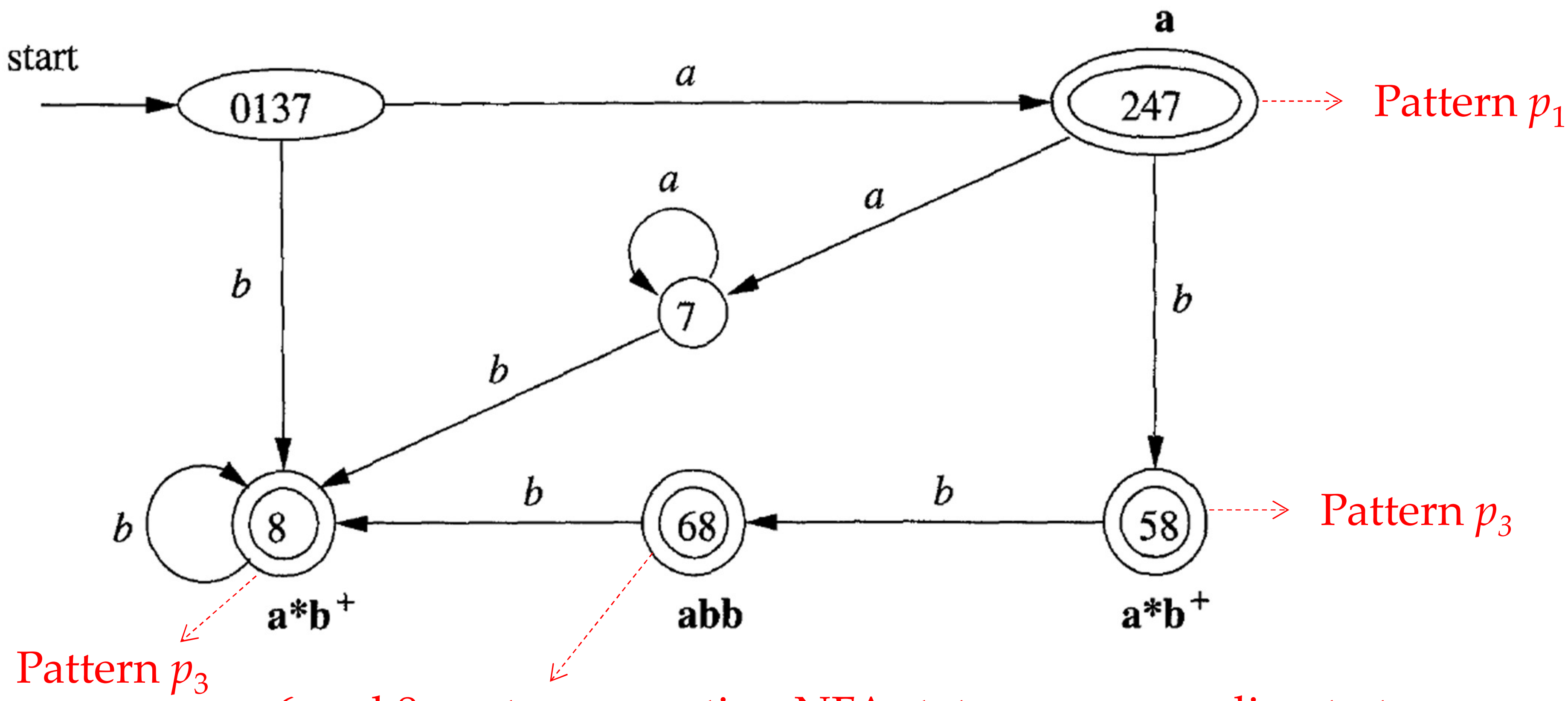
- Combining the three NFAs



Example

- Converting the big NFA to a DFA





6 and 8 are two accepting NFA states corresponding to two patterns. We choose Pattern p2, which is specified before p3

Reading Tasks

- Chapter 3 of the dragon book
  - 3.1 The role of the lexical analyzer
  - 3.3 Specification of tokens
  - 3.4 Recognition of tokens (lab content)
  - 3.5 The lexical-analyzer generator Lex (lab content)
  - 3.6 Finite automata
  - 3.7 From regular expressions to automata
  - 3.8 Design of a lexical analyzer generator
    - 3.8.1 – 3.8.3, the remaining can be skipped