编译原理
Complier Principles

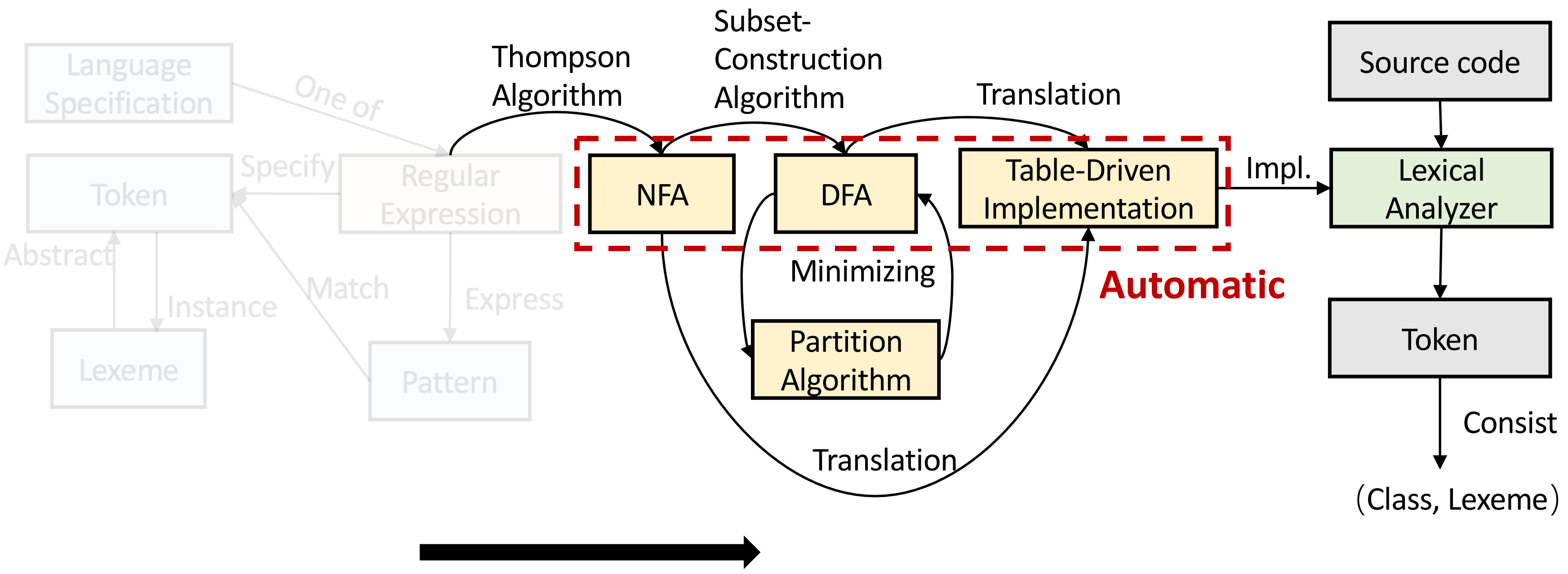
Lecture2
Lexical Analysis: NFA&DFA

赵帅

计算机学院

中山大学

Content



From Specification to Implementation

Finite Automata[有穷自动机]

- **REs is only a language specification**[只是定义了语言]
  - to construct a token recognizer for languages given by regular expressions
- **How do we go from specification to implementation?**
  - Regular expressions can be implemented using finite automata
  - There are two types of automata
    - NFAs (nondeterministic finite automata) [非确定的有穷自动机]
    - DFAs (deterministic finite automata) [确定的有穷自动机]

Finite Automata(FA) [有穷自动机]

Lexical Specification → Regular expressions → NFA → DFA → Table-driven Implementation

Finite Automata[有穷自动机]

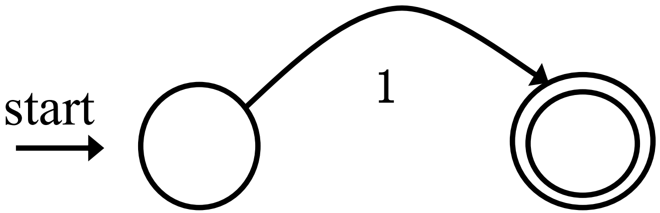
- **Regular Expression** = specification[正则表达是定义]
- **Finite Automata** = implementation[自动机是实现]

- Automaton (pl. automata): a machine or program
- **Finite automaton (FA):** a program with a finite number of states

- Finite Automata are similar to transition diagrams
  - They have states and labelled edges
  - There are one unique start state and one or more than one final states

Transition Diagram[转换图]

- **Node**[节点]**: state**
  - Each state represents a condition that may occur in the process
  - Initial state (Start): only one, circle marked with ‘start’
  - Final state (Accepting): may have multiple, double circle



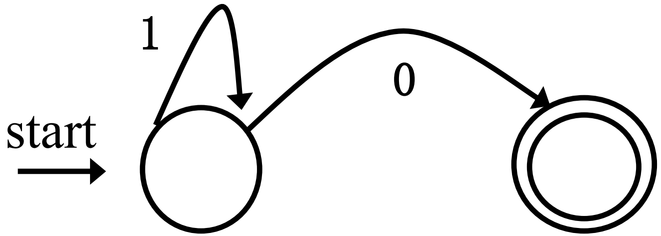
- **Edge**[边]**: transition. directed, labeled with the symbol(s)**
  - From one state to another on the input

FA: Language

- An FA is a program for classifying strings (return: accept, reject)
  - In other words, a program for recognizing a language
  - For a given string ‘x’, if there is a transition sequence for ‘x’ to move from the start state to a certain accepting state, then we say ‘x’ is accepted by the FA. Otherwise, rejected
- Language of FA = set of strings accepted by that FA
  - L(FA) ≡ L(RE)

Example

- Are the following strings acceptable?
  - 0 √
  - 1 ×
  - 11110 √
  - 11101 ×
  - 11100 ×
  - 1111110 √



- What language does the state graph recognize? ∑ = {0, 1}
  - Any number of '1's followed by a single 0

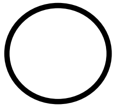
DFA and NFA

- Deterministic Finite Automata (DFA): the machine can exist in only one state at any given time[确定的有限状态机]
  - One transition per input per state
  - No ε-moves
  - Takes only one path through the state graph

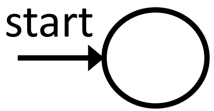
- Nondeterministic Finite Automata (NFA): the machine can exist in multiple states at the same time[非确定的有限状态机]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the end of input

State Graph

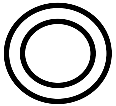
- 5 components (∑, S, n, F, δ)
  - An input alphabet Σ
  - A set of states S



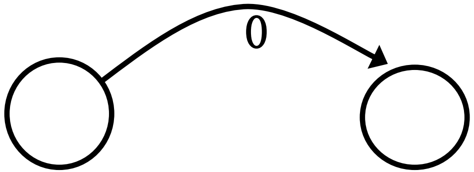
  - A start state n ∈ S



  - A set of accepting states F ⊆ S



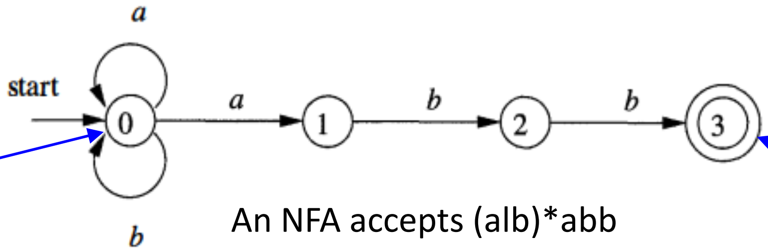
  - A set of transitions δ: $S_a \xrightarrow{\text{Input}} S_b$



Comparison of NFA and DFA

- NFA: There are many possible moves: to accept a string, we only need one sequence of moves that lead to a final state

Start state



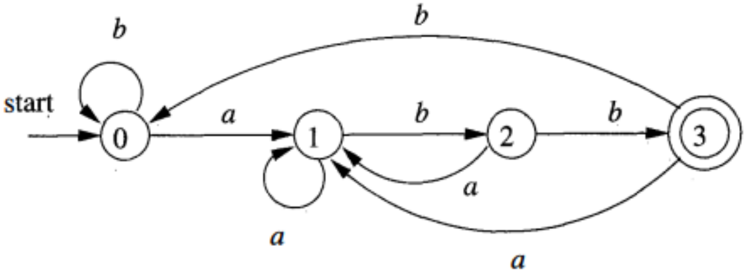
An NFA accepts (alb)*abb

Accepting state

  - Input string: aabb
  - Successful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$
  - Unsuccessful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

Comparison of NFA and DFA

- DFA: There is only one possible sequence of moves, either lead to a final state and accept or the input string is rejected

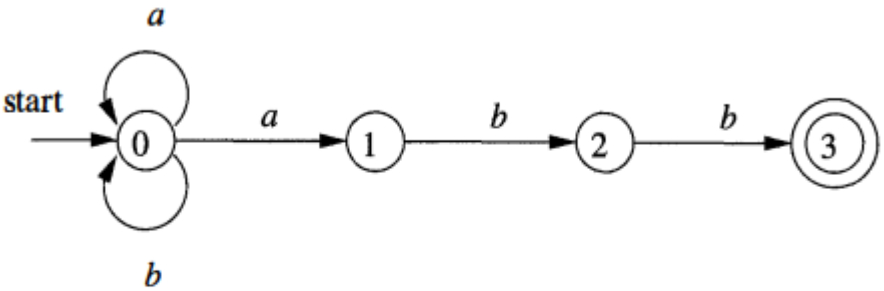


A DFA accepts (alb)*abb

  - Input string: aabb
  - Successful sequence: $0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Transition Table

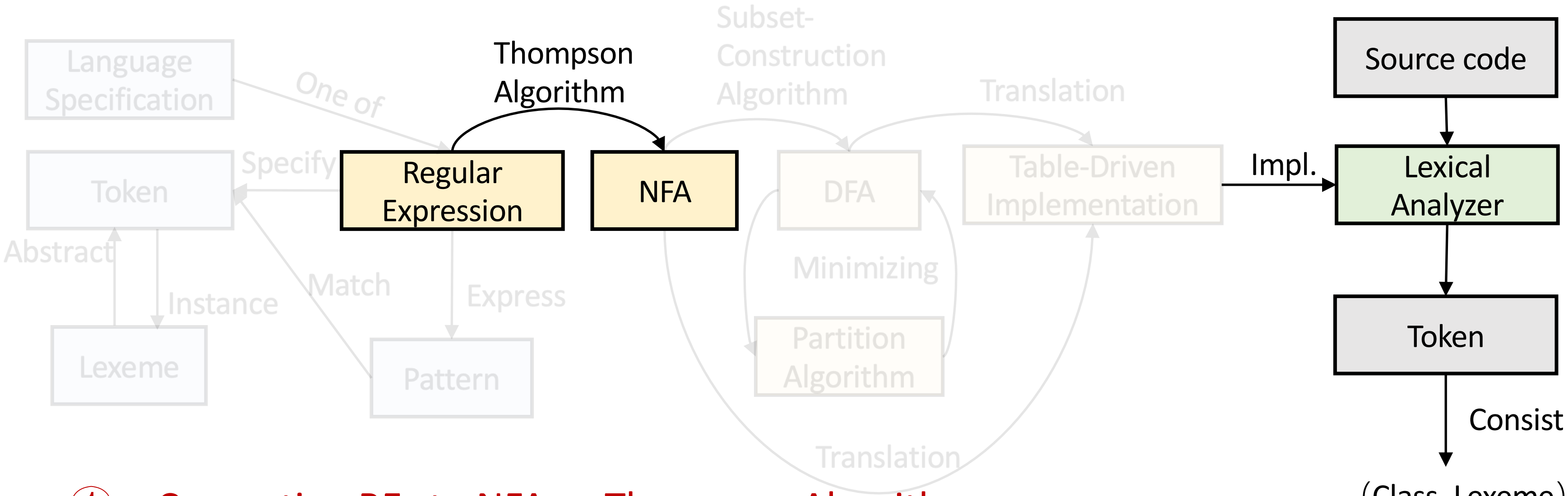
- FA can also be represented using **transition table**



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

- Advantage
  - We can easily find the transitions on a given state and input.
- Disadvantage
  - It takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols.
    - Need finite memory O(|S| * |∑|)

Content

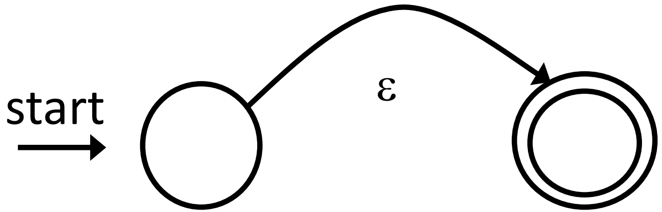


① Converting REs to NFAs – Thompson Algorithm
② Converting NFAs to DFAs
③ Perform DFA minimization
④ Converting DFAs to table-driven implementations

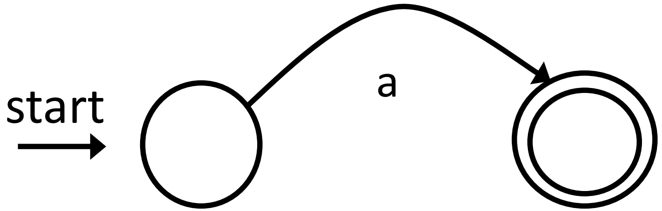
Construct NFA for RE

Basic: processing atomic REs （Thompson算法）

- **NFA for ε**



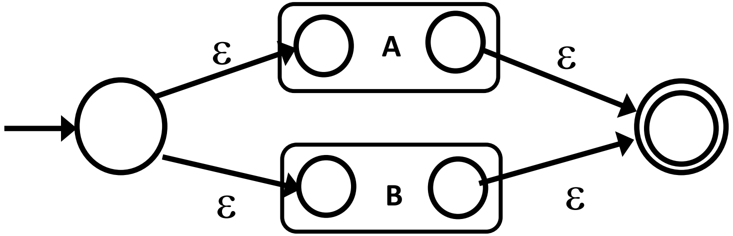
- **NFA for single character a**



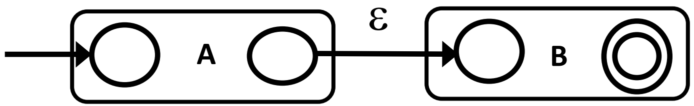
Construct NFA for RE

Inductive: processing compound Res

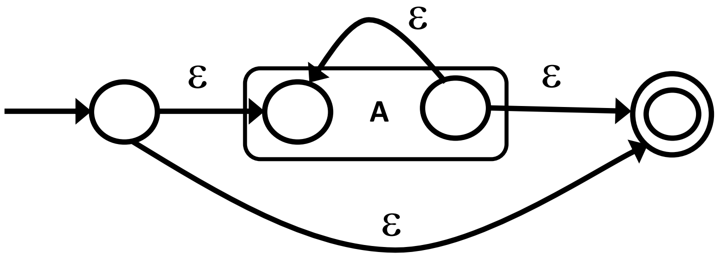
R=A|B



R=AB



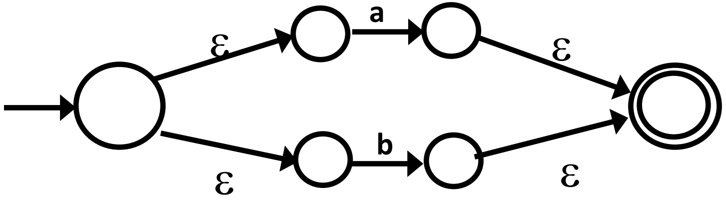
R=A*



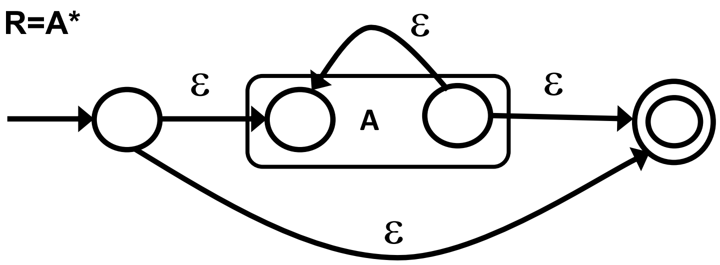
Example

- Convert “(a|b)*abb” to NFA

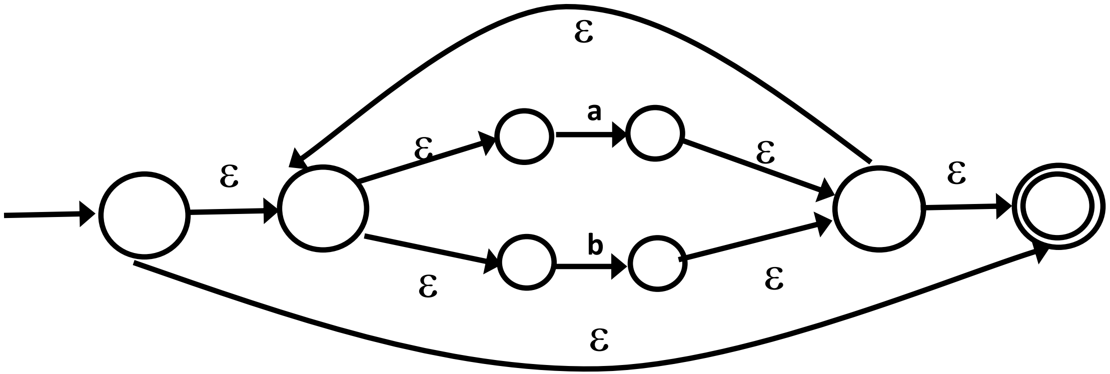
(a|b)



R=A*

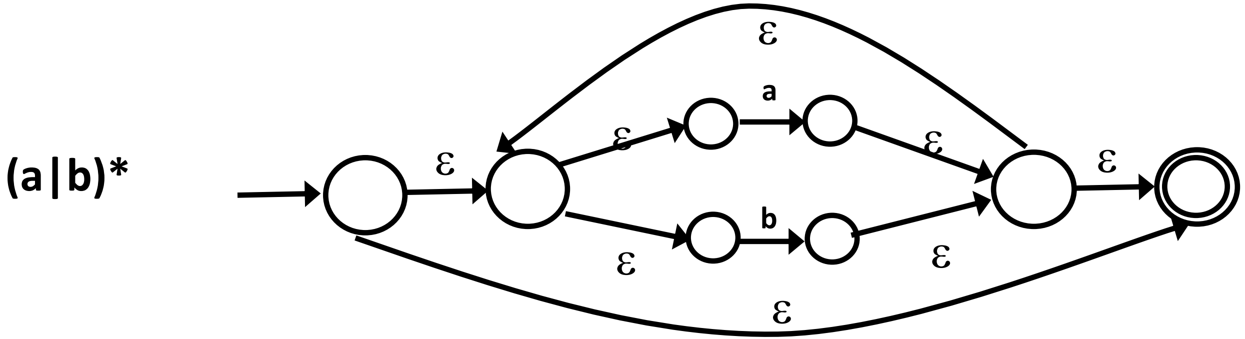


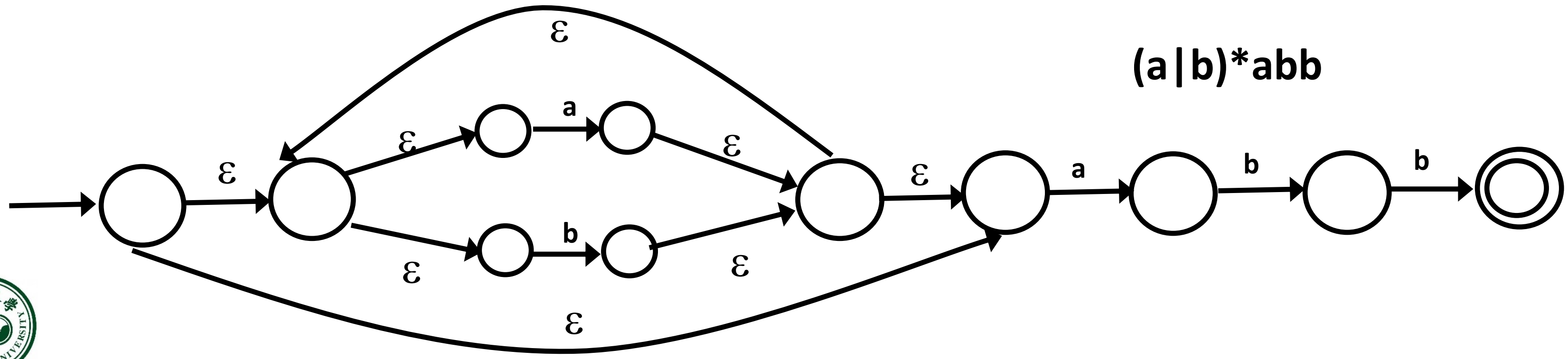
(a|b)*



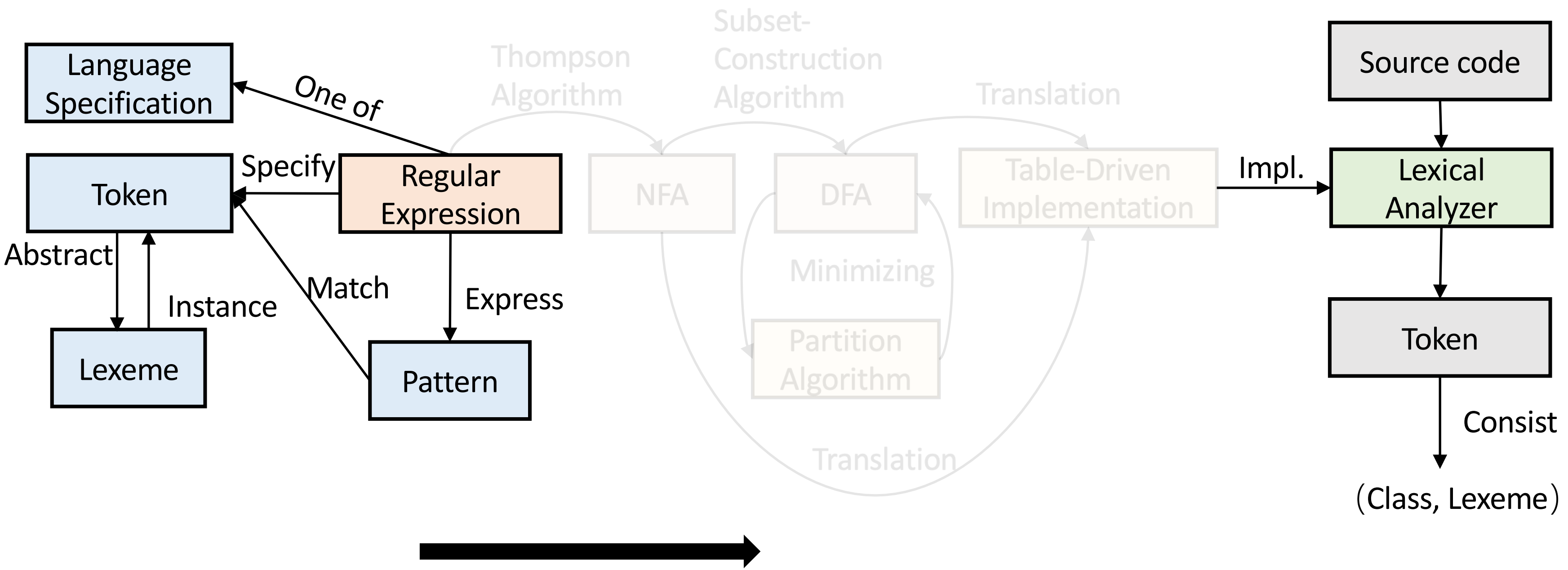
Example

- **Convert "(a|b)*abb" to NFA**



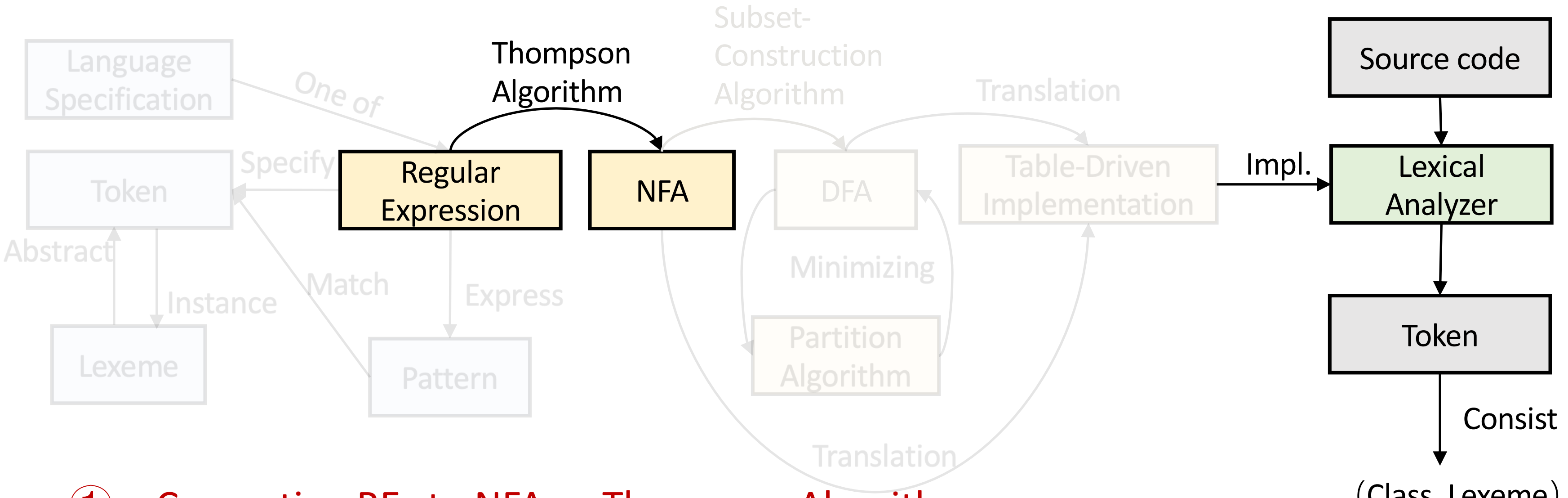


Revisit



From Specification to Implementation

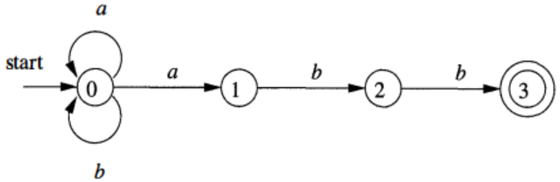
Revisit



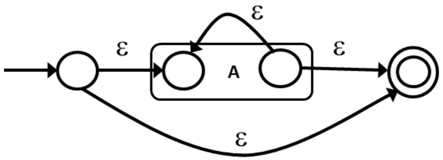
1. Converting REs to NFAs – Thompson Algorithm
2. Converting NFAs to DFAs – Subset-Construction Algorithm
3. Perform DFA minimization
4. Converting DFAs to table-driven implementations

Revisit

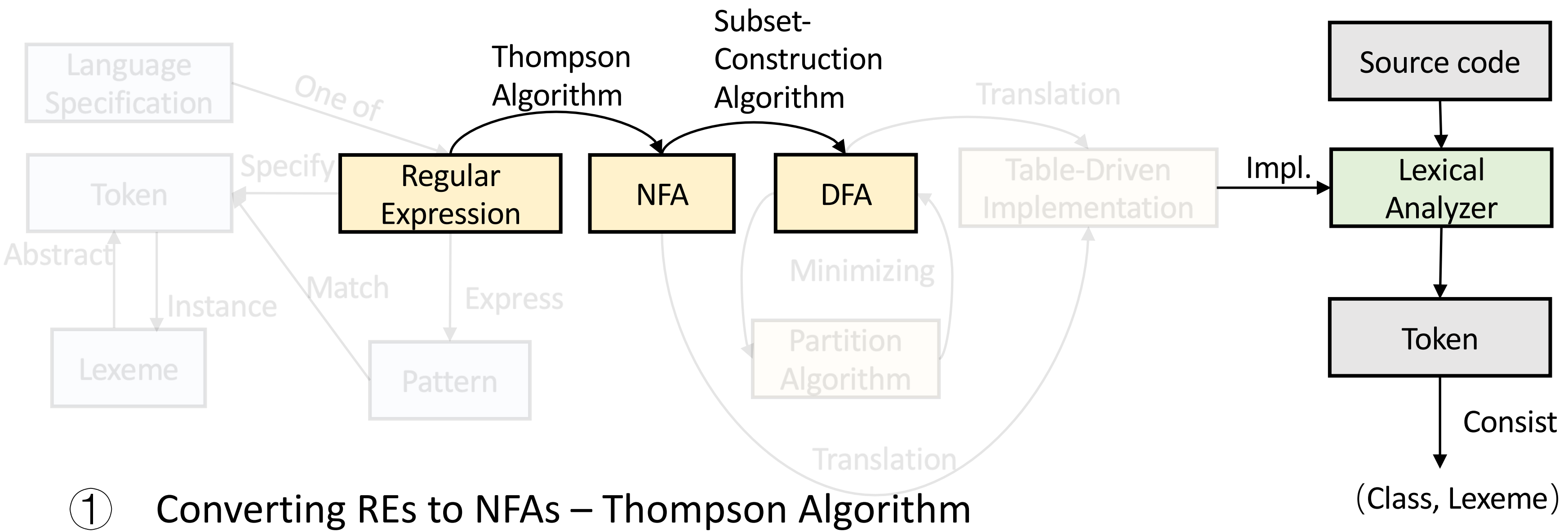
- Specification: Regular Expression
  - Compound Regular Expression
    - E.g., (ab)*, (a|b)*, (a*b*)*
  - Removing Ambiguity: keyword first, maximal match, the one listed first
- Implementation: Finite Automata
  - Non-Deterministic Finite Automata (NFA)
  - Deterministic Finite Automata (DFA)
  - Transition Table
  - Thompson Algorithm
    - From REs to NFAs (systemic way)
    - Use ε to connect small NFAs



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



Content



① Converting REs to NFAs – Thompson Algorithm

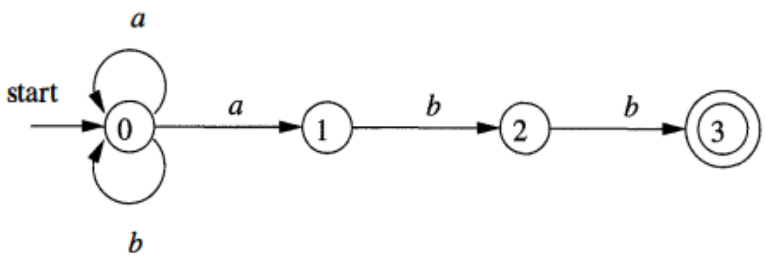
② Converting NFAs to DFAs – Subset-Construction Algorithm

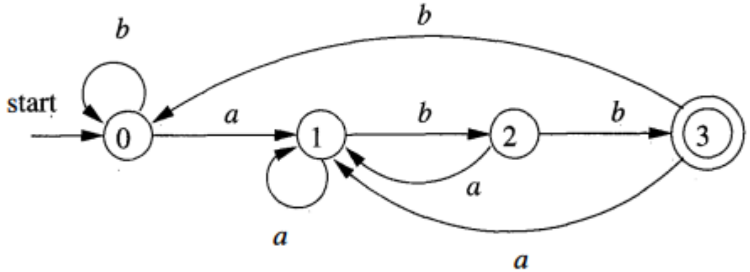
③ Perform DFA minimization

④ Converting DFAs to table-driven implementations

From NFA to DFA

- **NFA and DFA are equivalent**





To show this we must prove every DFA can be converted into an NFA which accepts the same language, and vice-versa

From NFA to DFA

- Theorem: L(NFA) ≡ L(DFA)
  - Both recognize regular languages L(RE)
  - 对于每个NFA M存在一个DFA M'，使得 L(M)=L(M’)
    - NFA与NFA描述能力相同!
- Resulting DFA consumes more memory than NFA
  - Potentially larger transition table as shown later
- But DFAs are faster to execute
  - For DFAs, number of transitions == length of input
  - For NFAs, number of potential transitions can be larger
  - NFA→ DFA conversion is needed because the speed of DFA far outweighs its extra memory consumption

From NFA to DFA

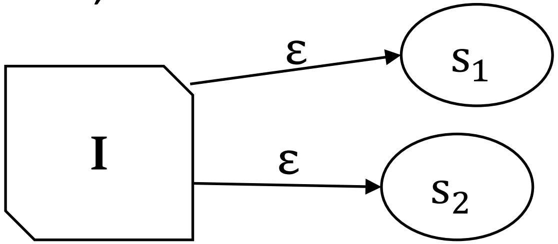
- Recall DFA
  - Every state must have exactly one transition defined for every letter
  - ε-moves are not allowed
    - NFAs have multiple transition, while DFAs can only have one transition in one time

- **Subset construction**[子集构造法]
  - **Each state of the constructed DFA corresponds to a set of NFA states**
    - After reading input $a_1 a_2 \ldots a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1 a_2 \ldots a_n$

From NFA to DFA

- Two problem need to solve
  - Eliminate ε-transition
  - Eliminate multiple transitions from a state on a single character

- **The ε-closure of a set of states**
  - The set of all states reachable by a series of zero or more ε-transitions from the **set of states**
  - That is, about **a set I**



ε-closure(I) = I∪$\{s_1, s_2\}$

From NFA to DFA: Algorithm

SUN YAT-SEN UNIVERSITY

Notion in the algorithm

- ε-closure(s)

  The set of all states reachable by a series of zero or more ε-transitions from **state s**

- ε-closure(T)

  The set of all states reachable by a series of zero or more ε-transitions from the **set of states T**

- $move(T,a) = \{t | s \in T \ and \ s \xrightarrow{a} t\}$

  Set of NFA states to which there is a transition on input symbol a from some state s in T

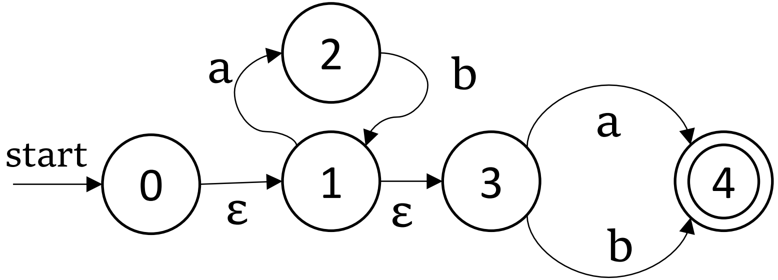
```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U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

Then, we will give a simple explanation by using the following symbols:

- I is a set of states,
- a is a character in the alphabet
- $move(I,a) = \{t | s \in I \ and \ s \xrightarrow{a} t\}$
- I_a = ε-closure($move(I,a)$)

Example

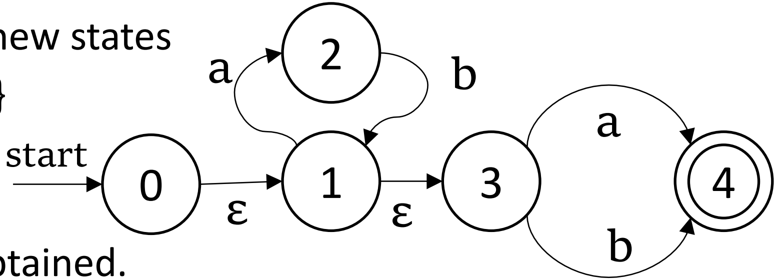
- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(state 0) = {0, 1, 3}
  - {0, 1, 3} is a new state for DFA, marked T0



I	I_a	I_b	Accept
{0, 1, 3} mark **T0**			

Example Cont.

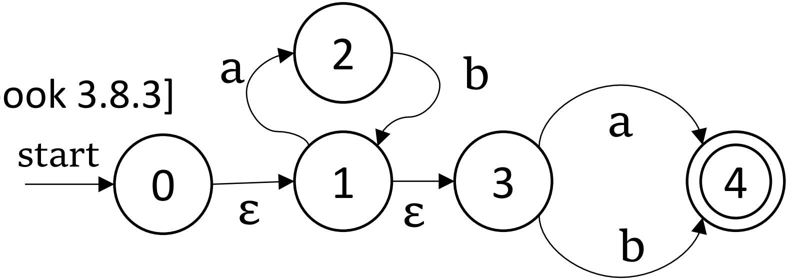
- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(state 0) = {0, 1, 3}
- Step2: Keep getting ε-closure(move(I, x)) for each character x in Σ
  - Computing for the new state until there are no more new states
  - I ={0, 1, 3}, move(I, a) = {2, 4}, ε-closure({2, 4}) = {2, 4}
  - I_a = ε-closure(move(I, a)) = {2, 4}
  - I_b = ε-closure(move(I, b)) = {4}
  - {2, 4} and {4} are sets of state that have never been obtained.



I	I_a	I_b	Accept
{0, 1, 3} mark **T0**	{2, 4} mark **T1**	{4} mark **T2**	
{2, 4} **T1**			
{4} **T2**			

Example Cont.

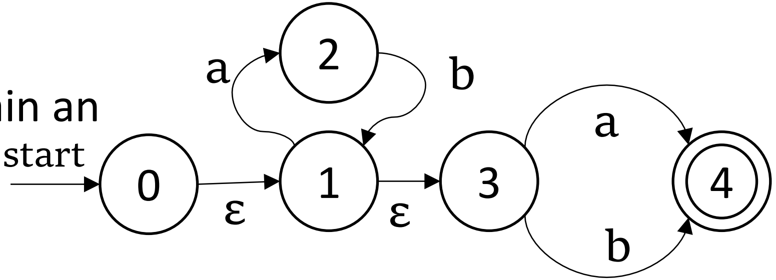
- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(state 0) = {0, 1, 3}
- Step2: Keep getting ε-closure(move(I, x)) for each character x in Σ
  - For T1 = {2, 4}
  - I_a = ε-closure(move({2, 4} , a)) = {} dead state [text book 3.8.3]
  - I_b = ε-closure(move({2, 4} , b)) = {1, 3} new state
  - For T2 = {4}, ….
  - Stop, when there are no more new states



I	I_a	I_b	Accept
{0, 1, 3} mark **T0**	{2, 4} mark T1	{4} mark T2	
{2, 4} **T1**		{1, 3} mark T3	
{4} **T2**			
{1,3} **T3**	{2,4} T1	{4} T2	

Example Cont.

- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(state 0) = {0, 1, 3}
- Step2: Keep getting ε-closure(move(I, x)) for each character x in Σ
  - Stop, when there are no more new states
- Step3: Mark as accepting for those states that contain an accepting state



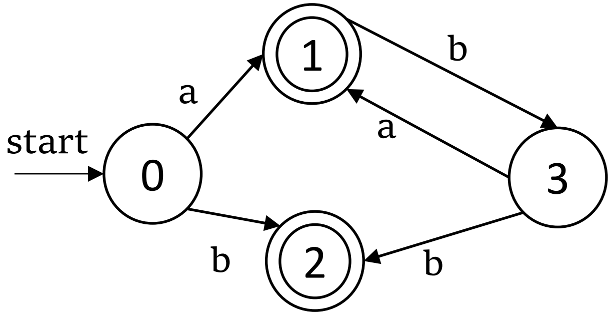
I	I_a	I_b	Accept
{0, 1, 3} **mark T0**	{2, 4} mark T1	{4} mark T2	**T0** No
{2, 4} **T1**		{1, 3} mark T3	**T1** Yes
{4} **T2**			**T2** Yes
{1,3} **T3**	{2,4} T1	{4} T2	**T3** No

Example Cont.

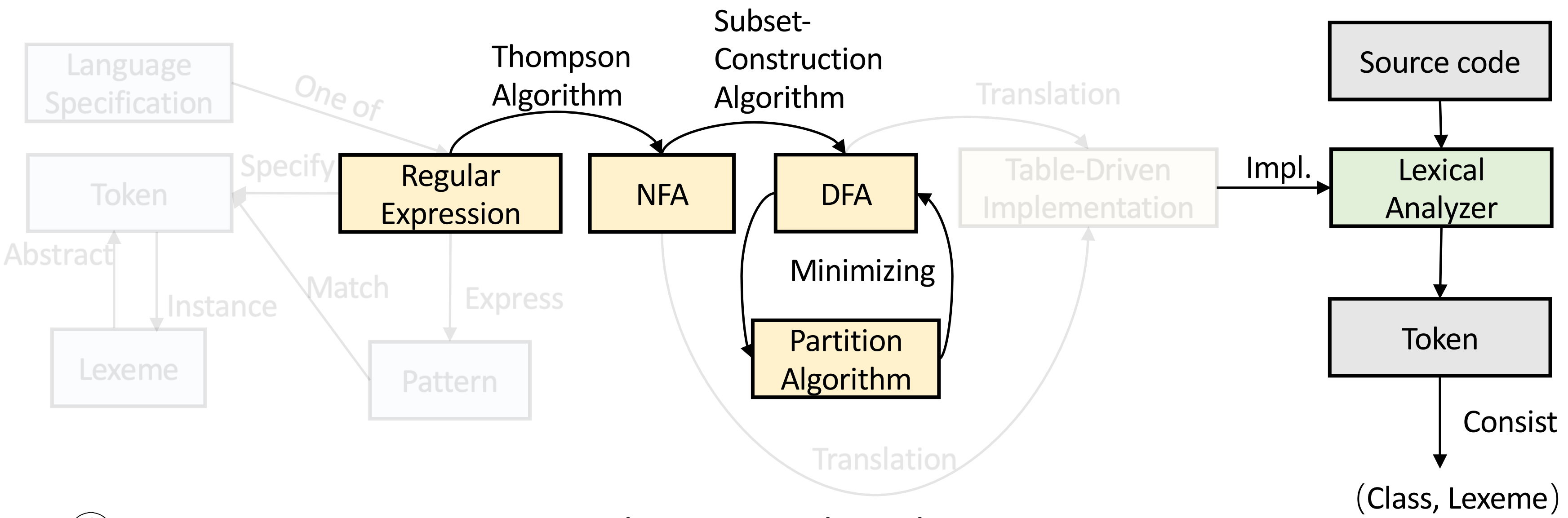
SUN YAT-SEN UNIVERSITY

- **Construct DFA**

I	I_a	I_b	Accept
{0, 1, 3} T0	{2, 4} mark T1	{4} mark T2	T0 No
{2, 4} T1		{1, 3} mark T3	T1 Yes
{4} T2			T2 Yes
{1,3} T3	{2,4} T1	{4} T2	T3 No



Content



① Converting REs to NFAs – Thompson Algorithm
② Converting NFAs to DFAs – Subset-Construction Algorithm
③ Perform DFA minimization – Partition Algorithm
④ Converting DFAs to table-driven implementations

Minimizing DFA

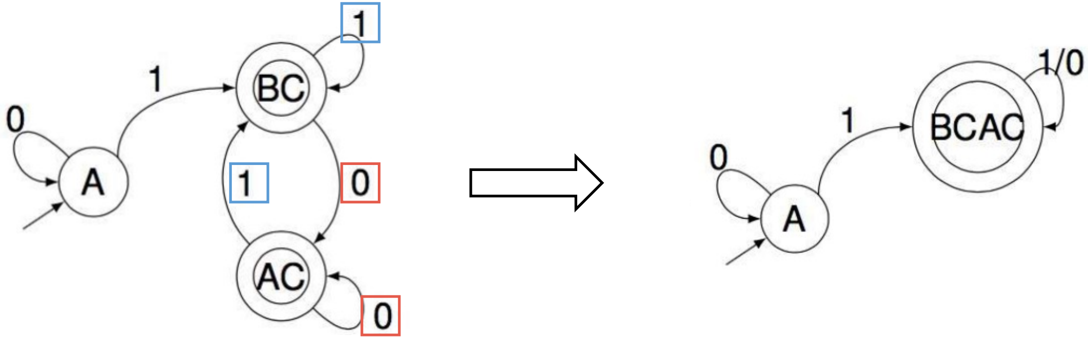
- **Theory:** Given any DFA, there is an equivalent DFA containing a minimum number of states, and this minimum-state DFA is unique

- **Equivalent States**

  If s and t are two states, they are equivalent if and only if:

  ① s and t are both accepting states or both non-accepting states.

  ② For each character $x \in \Sigma$, s and t have transitions on x to the equivalent states



States BC and AC do not need differentiation

Minimization Algorithm

- **The algorithm**

  Partitioning the states of a DFA into groups of states that cannot be distinguished (i.e., equivalent)

① First, split the set of states into two sets, one consists of all accepting states and the other consists of all non-accepting states.

② Consider the transitions on each character ‘x’ of the alphabet for each subset, and determine whether all the states in the subset are equivalent, or the subset should be split.

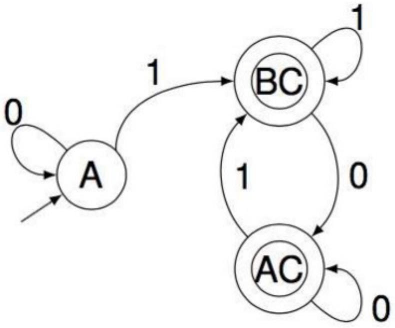
③ Continue this process until no further splitting of sets occurs

Simple Example for Minimizing DFA

- **Step 1： Divide the states into two sets**

  Initial sets: {non-accepting states}, {accepting states}

  Initial: {A} , {BC, AC}



- **Step 2:   check if the states are equivalent**

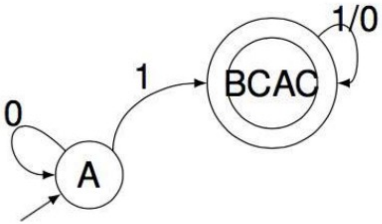
  For {BC, AC}

  BC on ‘0’ → AC, AC on ‘0’ → AC

  BC on ‘1’ → BC, AC on ‘1’ → BC

  No way to distinguish BC from AC on any string starting with ‘0’ or ‘1’

  Final: {A}, {BCAC}



Example: Minimization

① **Initial**

{S, A, B} and {C, D, E, F}, {non-accepting states} and {accepting states}

② **Consider all states in each subset, check the transitions for each x∈Σ**

Now we have two subsets {C,D,E,F} and {S,A,B}

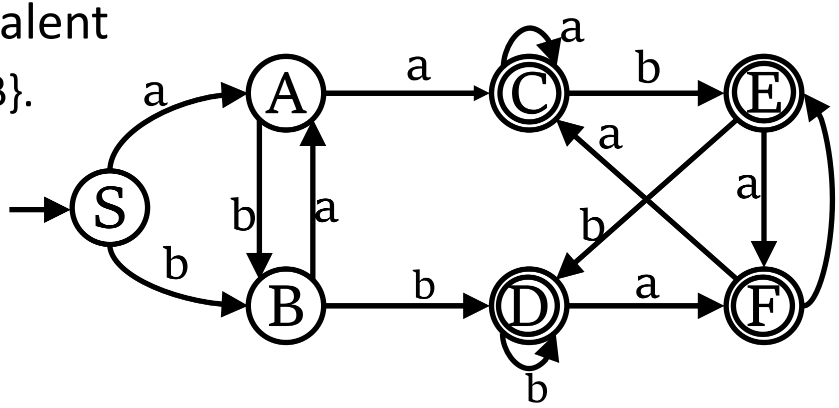
For I_1 = {C, D, E, F}

$$\left.\begin{array}{l}\{C,D,E,F\} \xrightarrow{a} \{C,F\} \Rightarrow \{C,D,E,F\} \xrightarrow{a} \{C,D,E,F\} \\ \{C,D,E,F\} \xrightarrow{b} \{D,E\} \Rightarrow \{C,D,E,F\} \xrightarrow{b} \{C,D,E,F\}\end{array}\right\} \Rightarrow$$

For each character x ∈{a, b}, all the states in {C, D, E, F} have the same transition on x.

All the states in {C, D, E, F} are equivalent

Now we still have {C,D,E,F} and {S,A,B}.



Example: Minimization Cont.

① **Initial**

{S, A, B} and {C, D, E, F}

② **Consider all states in each subset, check the transitions for all x∈Σ**

For I_1 = {C, D, E, F}, all states in I_1 are equivalent.

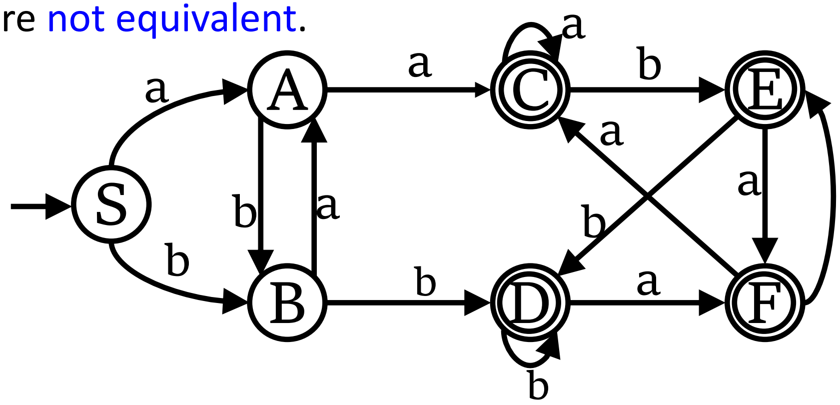
For I_2 = {S, A, B}, for each states, check x ∈{a, b}

Check character a

$\{S\} \xrightarrow{a} \{A\}, \{A\} \xrightarrow{a} \{C\}, \{B\} \xrightarrow{a} \{A\}$ ⇒ $\{S, B\} \xrightarrow{a} \{A\}, \{A\} \xrightarrow{a} \{C, D, E, F\}$ ⇒ {S, B} and A have different transition on a ⇒ {S, B} and A are not equivalent.

So split {S, A, B} to {S, B} and {A}.

Now we have {C, D, E, F}, {S, B}, {A}.



Example: Minimization Cont.

① **Initial**

{S, A, B} and {C, D, E, F}

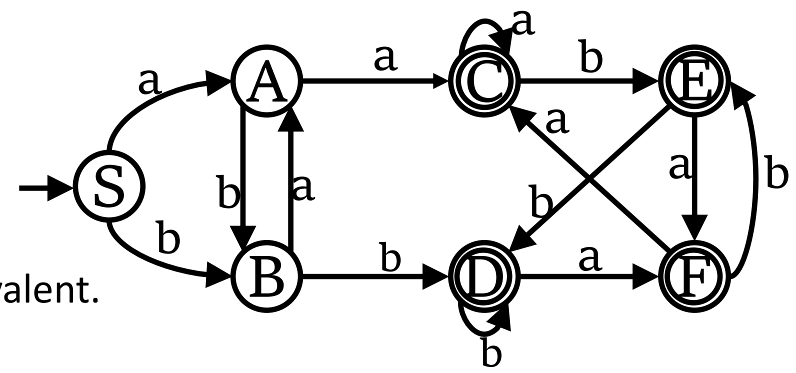
② **Consider all states in each subset, check the transitions for all x∈Σ**

For I_1 = {C, D, E, F}, all states in I_1 are equivalent. Now we still have {C,D,E,F} and {S,A,B}.

For I_2 = {S, A, B}, for each states, check x ∈{a, b}

Now we have {C, D, E, F}, {S, B}, {A}.

Keep checking the subset {S, B}.



$$\{S,B\} \xrightarrow{a} \{A\}$$

$\{S\} \xrightarrow{b} \{B\}, \{B\} \xrightarrow{b} \{C,D,E,F\}$ ⇒ S and B are not equivalent.

So split {S, B} to {S} and {B}.

Now we have {C,D,E,F}, {S}, {B}, {A}.

Since all states in {C,D,E,F} are equivalent and {S},{B},{A} each contain only one state, no further splitting of sets will occur. Stop this process.

Example: Minimization Cont.

③ Finally, get the subsets and draw min DFA

{C, D, E, F}, {S}, {A}, {B}

{C, D, E, F} is denoted as {C}

$\{S\} \xrightarrow{a} \{A\}$

$\{S\} \xrightarrow{b} \{B\}$

$\{A\} \xrightarrow{a} \{C,D,E,F\}$

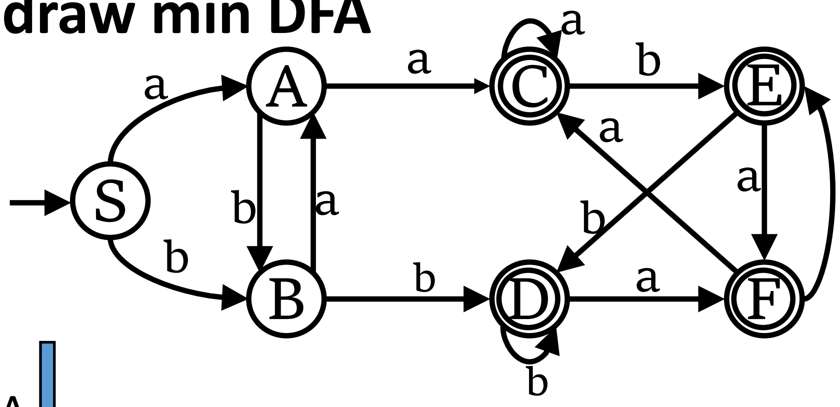
$\{A\} \xrightarrow{b} \{B\}$

$\{B\} \xrightarrow{a} \{A\}$

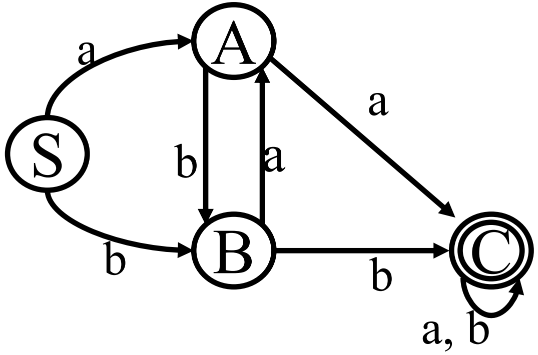
$\{B\} \xrightarrow{b} \{C,D,E,F\}$

$\{C,D,E,F\} \xrightarrow{a} \{C,D,E,F\}$

$\{C,D,E,F\} \xrightarrow{b} \{C,D,E,F\}$



Minimization DFA



Example

- **Is the DFA minimal?**

  1. Initial: {0,3} and {1,2}
  2. Check all states in {1,2}

     state 2 have no transition,

     state 1 have transition on b.

     {1} and {2} are not equivalent.

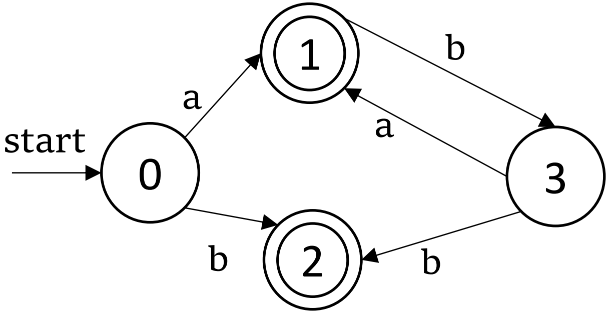
     Now we have {0,3} {1} {2}.
  3. Check all states in {0,3}

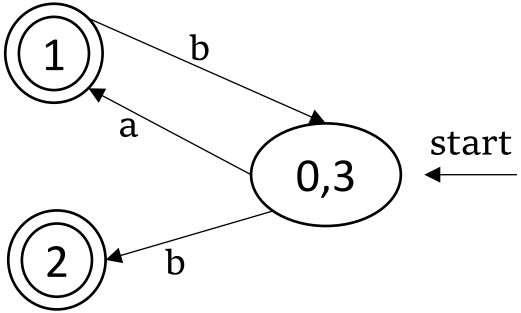
     Move({0,3}, a) = {1}

     Move({0,3}, b) = {2}.

     0 and 3 are equivalent states.

  Result: {0,3} {1} {2}.





NFA → DFA: Space Complexity[复杂度]

- NFA may be in many states at any time
- How many different possible states in DFA?
  - If there are N states in NFA, the DFA must be in some subset of those N states
  - How many non-empty subsets are there?
    - $2^N - 1$
- The resulting DFA has $O(2^N)$ space complexity, where N is number of original states in NFA
  - For real languages, the NFA and DFA have about same number of states

NFA → DFA: Time Complexity[复杂度]

SUN YAT-SEN UNIVERSITY

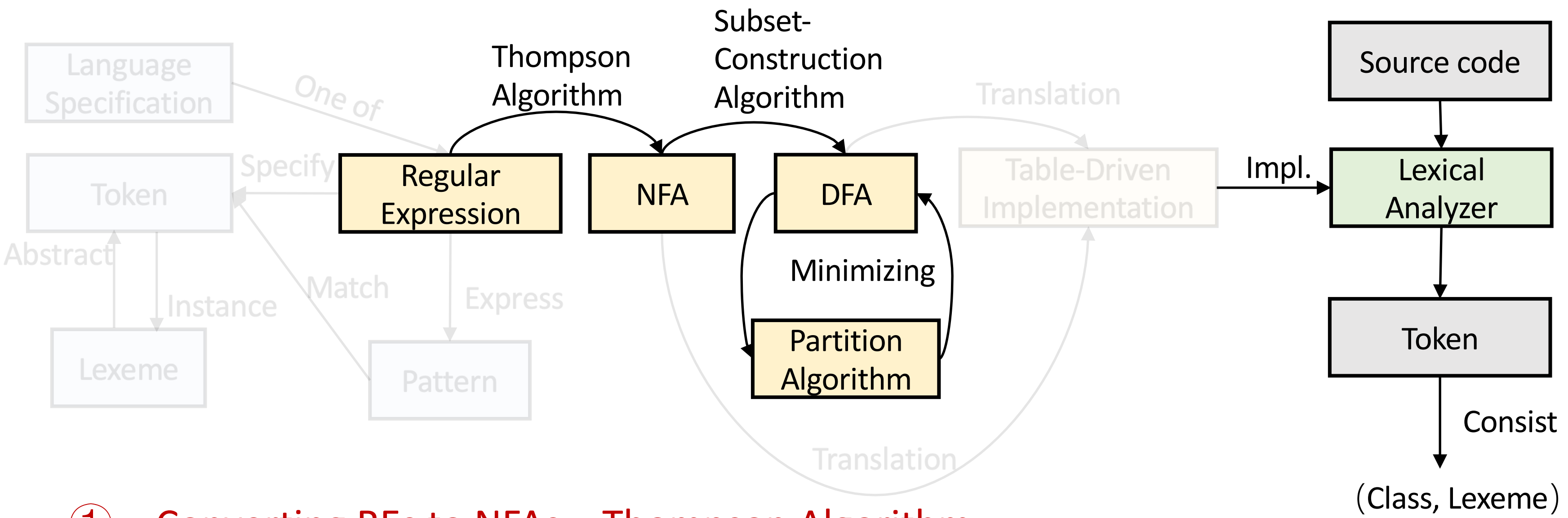
- DFA execution
  - Requires O(|X|) steps, where |X| is the input length
  - Each step takes constant time
    - If current state is S and input is c, then read T[S, c]
    - Update current state to state T[S, c]
  - Time complexity = O(|X|)

Deterministic:
For input c,
unique transition

- NFA execution
  - Requires O(|X|) steps, where |X| is the input length
  - Each step takes O(N^2) time, where N is the number of states
    - Current state is a set of potential states, up to N
    - On input c, must union all T[$S_{potential}$, c], up to N times
      - Each union operation takes O(N) time
  - Time complexity = O(|X|* N^2)

Non-deterministic:
For input c,
form current state,
your can transit to any
states (including itself)

Revisit

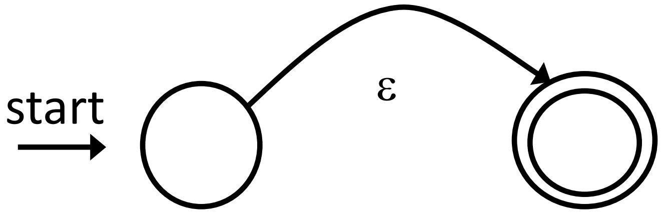


① Converting REs to NFAs – Thompson Algorithm
② Converting NFAs to DFAs – Subset-Construction Algorithm
③ Perform DFA minimization – Partition Algorithm
④ Converting DFAs to table-driven implementations

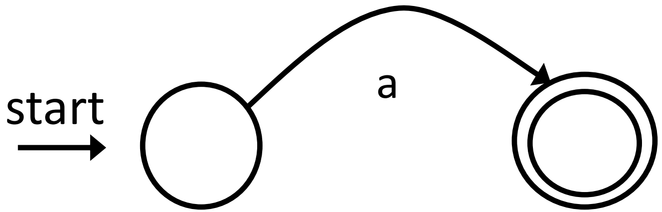
Construct NFA for RE (Revisit)

Basic: processing atomic REs （Thompson算法）

- **NFA for ε**



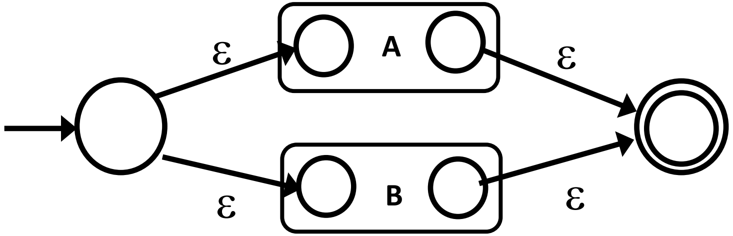
- **NFA for single character a**



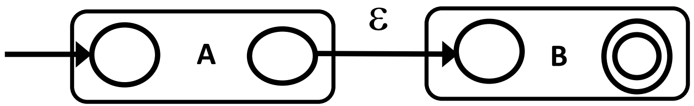
Construct NFA for RE (Revisit)

Inductive: processing compound Res

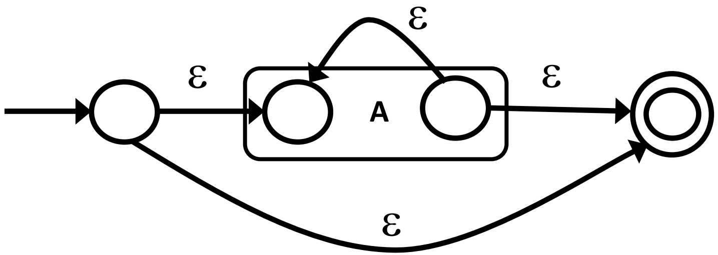
R=A|B



R=AB



R=A*



From NFA to DFA (Revisit)

SUN YAT-SEN UNIVERSITY

Notion in the algorithm

- ε-closure(s)

  The set of all states reachable by a series of zero or more ε-transitions from **state s**

- ε-closure(T)

  The set of all states reachable by a series of zero or more ε-transitions from the **set of states T**

- $move(T,a) = \{t | s \in T \ and \ s \xrightarrow{a} t\}$

  Set of NFA states to which there is a transition on input symbol a from some state s in T

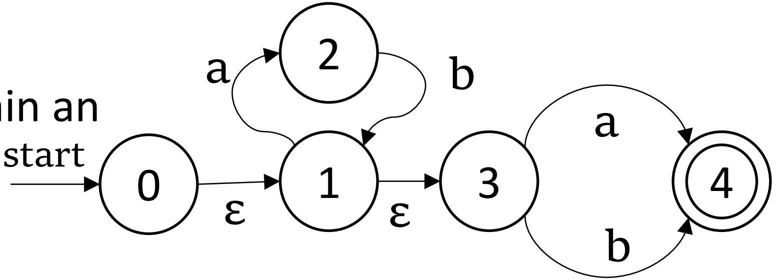
```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T,a));
                if ( U is not in Dstates )
                        add U as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

Then, we will give a simple explanation by using the following symbols:

- I is a set of states,
- a is a character in the alphabet
- $move(I,a) = \{t | s \in I \ and \ s \xrightarrow{a} t\}$
- I_a = ε-closure($move(I,a)$)

From NFA to DFA (Revisit)

- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(state 0) = {0, 1, 3}
- Step2: Keep getting ε-closure(move(I, x)) for each character x in Σ
  - Stop, when there are no more new states
- Step3: Mark as accepting for those states that contain an accepting state



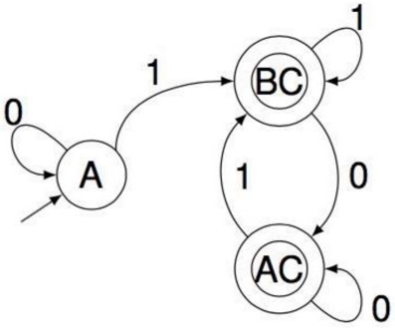
I	I_a	I_b	Accept
{0, 1, 3} mark **T0**	{2, 4} mark T1	{4} mark T2	**T0** No
{2, 4} **T1**		{1, 3} mark T3	**T1** Yes
{4} **T2**			**T2** Yes
{1,3} **T3**	{2,4} T1	{4} T2	**T3** No

Minimizing DFA (Revisit)

- **Step 1： Divide the states into two sets**

  Initial sets: {non-accepting states}, {accepting states}

  Initial: {A} , {BC, AC}



- **Step 2: check if the states are equivalent**

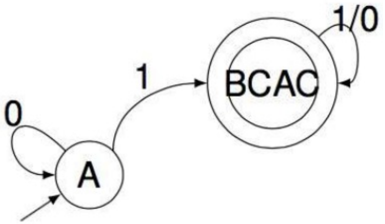
  For {BC, AC}

    BC on ‘0’ → AC, AC on ‘0’ → AC

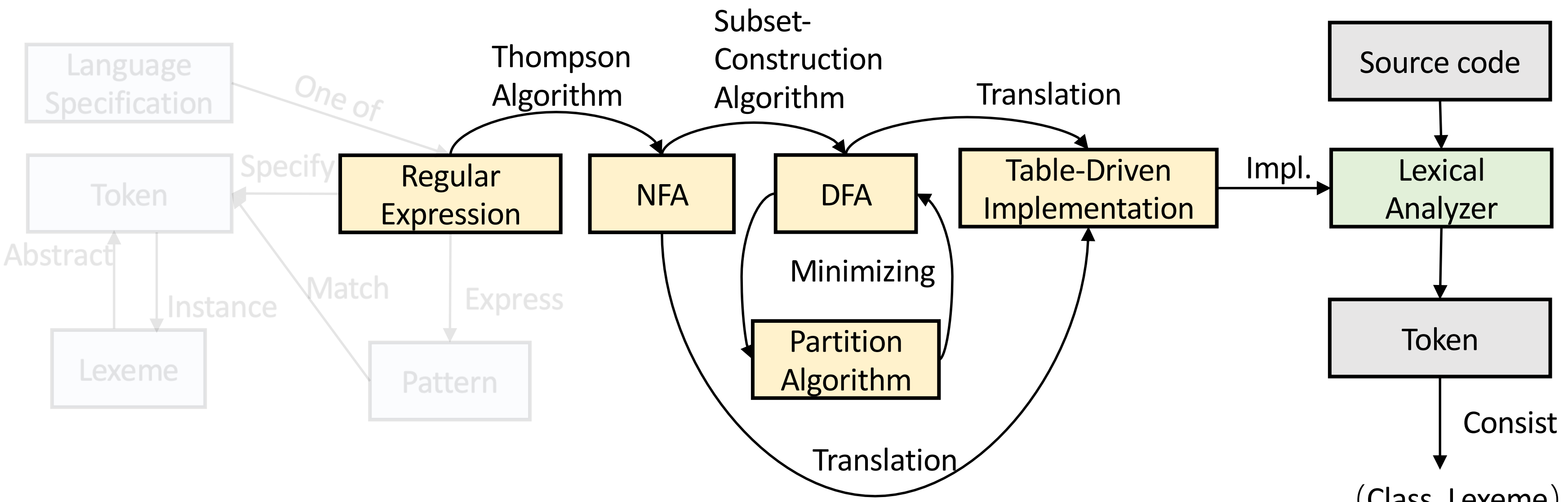
    BC on ‘1’ → BC, AC on ‘1’ → BC

    No way to distinguish BC from AC on any string starting with ‘0’ or ‘1’

  Final: {A}, {BCAC}



Content



① Converting REs to NFAs – Thompson Algorithm

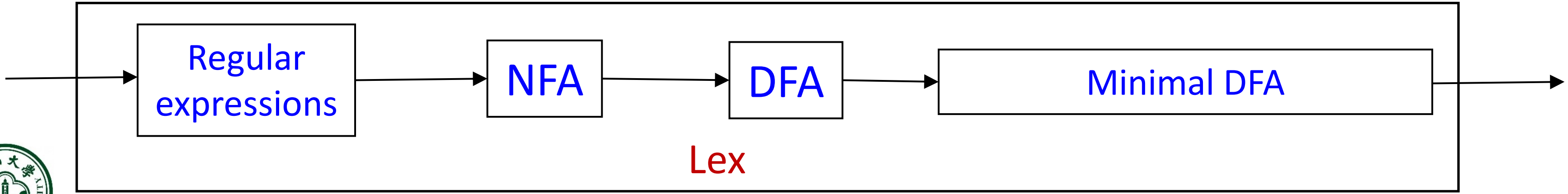
② Converting NFAs to DFAs – Subset-Construction Algorithm

③ Perform DFA minimization – Partition Algorithm

④ Converting DFAs to table-driven implementations

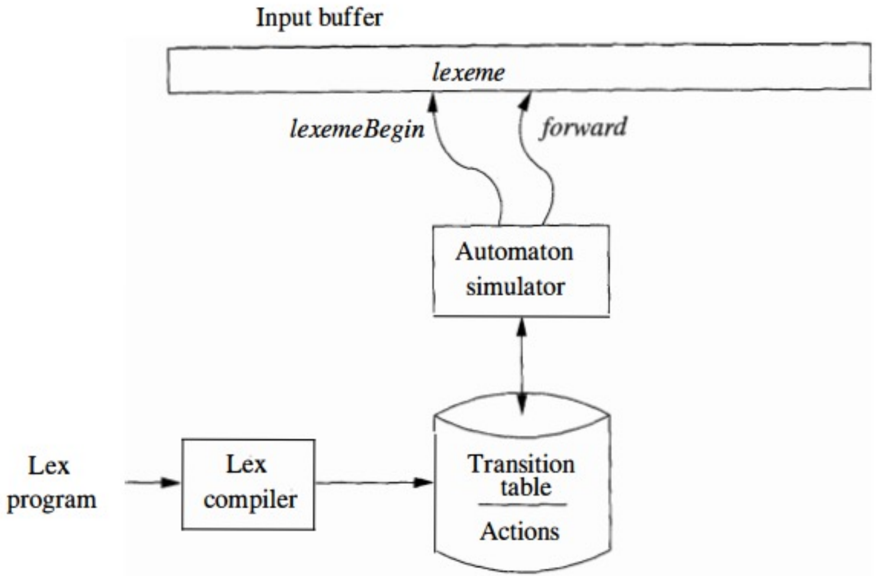
Implementation in Practice[实际实现]

- **Lex[词法分析器]: RE → NFA → DFA → Table**
  - Converts regular expressions to NFA
  - Converts NFA to DFA
  - Performs DFA state minimization to reduce space
  - Generate the transition table from DFA
  - Performs table compression to further reduce space
- Most other automated lexers also choose DFA over NFA
  - Trade off space for speed



Lexical Analyzer Generated by Lex

- A Lex program is turned into a transition table and actions, which are used by a FA simulator
- Automaton need to recognize lexemes matching any of the patterns in a program

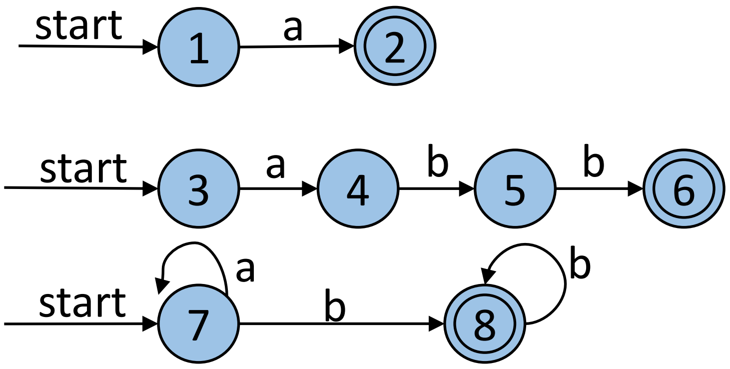


STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Lex: Example

- Three patterns, three NFAs

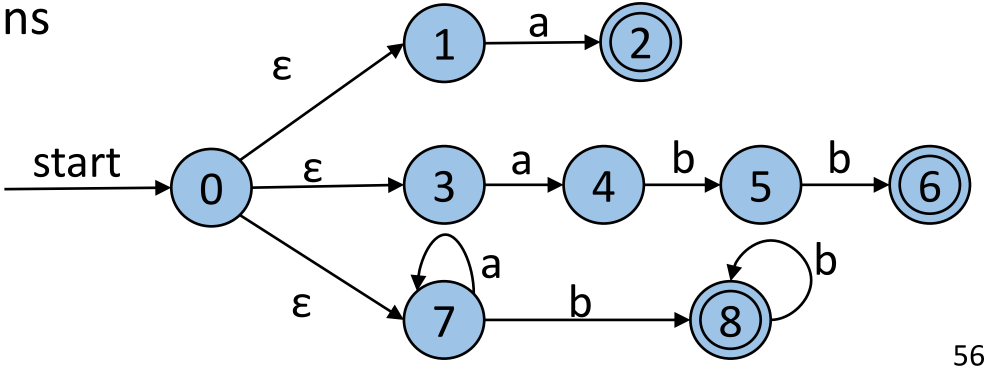
a	{action$_1$}
abb	{action$_2$}
a*b+	{action$_3$}



- Combine three NFAs into a single NFA

  Add start state 0 and ε-transitions

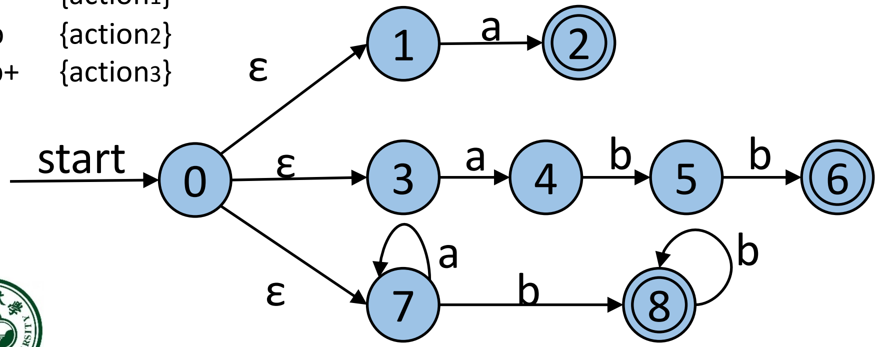
Any one is possible,
if you haven't read
any input symbol

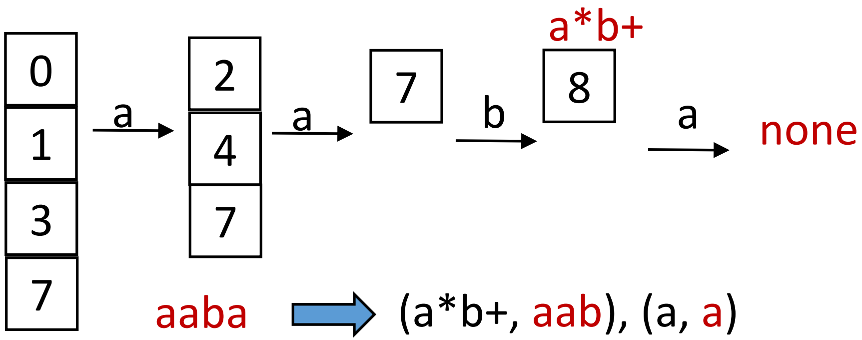


Lex: Example

- Input: aaba
  - ε-closure(0) = {0, 1, 3, 7}
  - Empty states after reading the fourth input symbol
    - There are no transitions out of state 8
    - Back up, looking for a set of states that include an accepting state
  - State 8: a*b+ has been matched
  - Select aab as the lexeme, execute action3
    - Return to parser indicating that token with pattern a*b+ has been found

a	{action1}
abb	{action2}
a*b+	{action3}

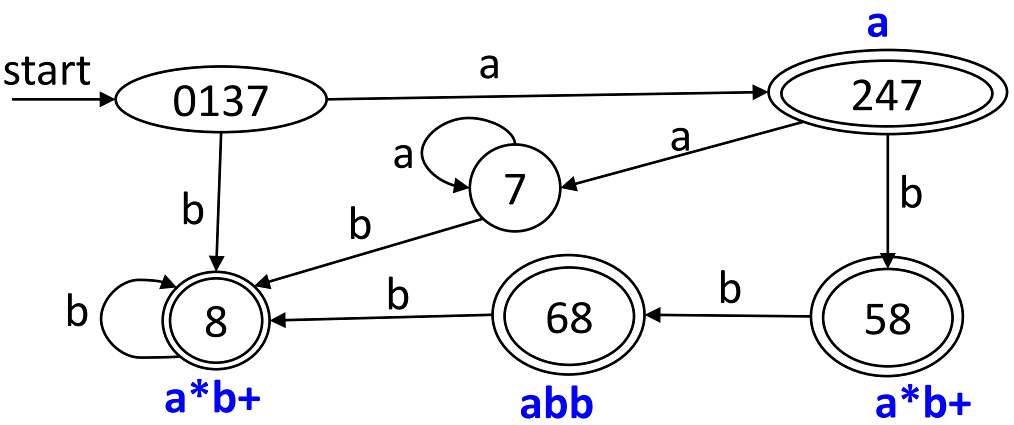




aaba ⇒ (a*b+, aab), (a, a)

Lex: Example

- DFA's for lexical analyzer
- Input: abba
  - Sequence of states entered: 0137 → 247 → 58 → 68
  - At the final a, there is no transition out of state 68
    - 68 itself is an accepting state that reports pattern abb



How Much Should We Match?[匹配多少]

- In general, find the longest match possible
  - We have seen examples
  - One more example: input string aabbb …
    - Have many prefixes that match the third pattern
    - Continue reading b’s until another a is met
    - Report the lexeme to be the initial a’s followed by as many b’s as there are
- If same length, appearing first takes precedence[先出现的优先]
  - String abb matches both the second and third pattern
  - We consider it as a lexeme for pattern2, since that pattern listed first

1	a	$\{action_1\}$
2	abb	$\{action_2\}$
3	a*b+	$\{action_3\}$

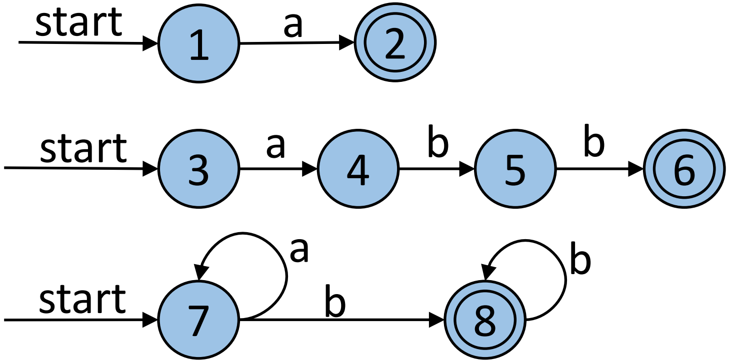
How to Match Keywords?[匹配关键字]

- Example: to recognize the following tokens
  - Identifiers: letter(letter | digit)*
  - Keywords: if, then, else
- **Approach 1**: make REs for keywords and place them before REs for identifiers so that they will take precedence
  - Will result in a more bloated[臃肿] finite state machine
- **Approach 2**: recognize keywords and identifiers using the same RE but differentiate using special keyword table
  - Will result in more streamlined finite state machine
  - But extra table lookup is required
- Usually approach 2 is more efficient than 1

Lex: Example

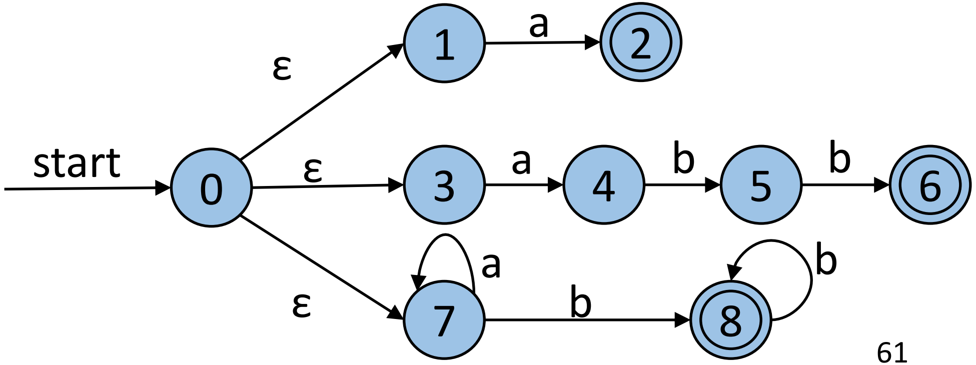
- Three patterns, three NFAs

a	{**action$_1$**}
abb	{**action$_2$**}
a*b+	{**action$_3$**}



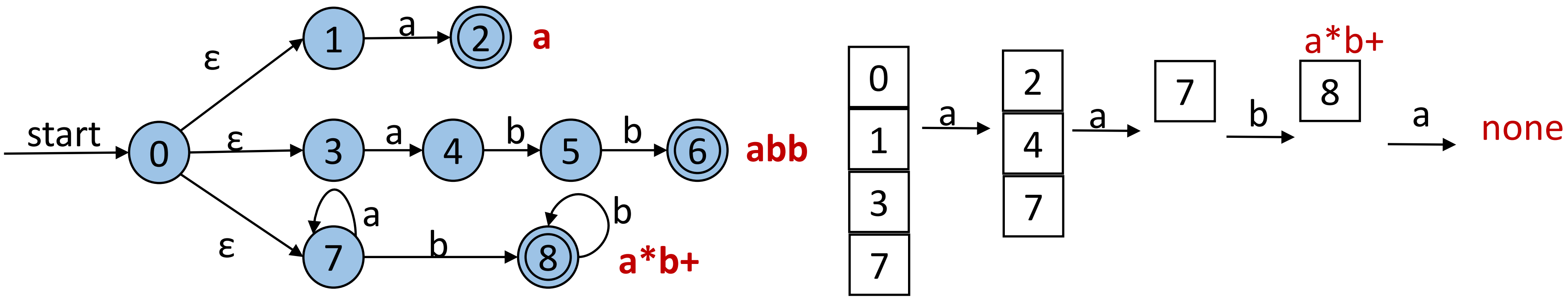
- Combine three NFAs into a single NFA

  Add start state 0 and ε-transitions



Lex: Example

- Input: abbb
  - ε-closure(0) = {0, 1, 3, 7}
  - Select aab as the lexeme, execute {action3}
    - Return to parser indicating that token with pattern a*b+ has been found



Q: 这个例子中，有几处二义性问题?

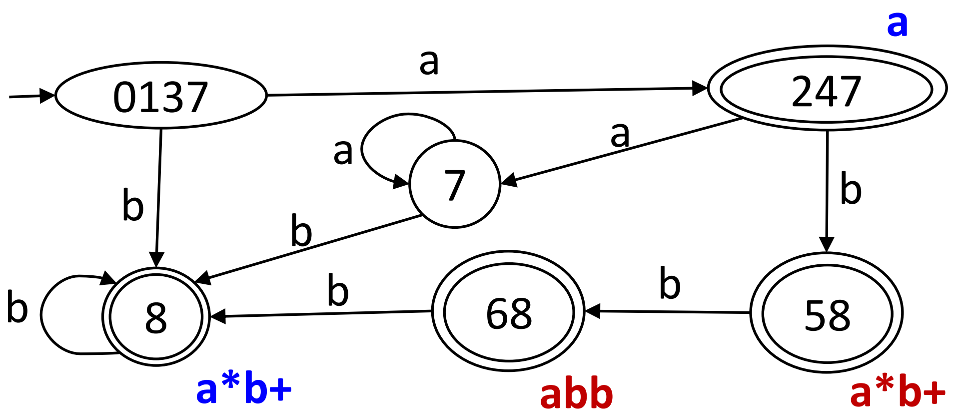
Why not a?

Why not a*b+?[二义性问题]

Lex: Example

- The accepting states are labeled by the pattern that is identified by that state.
  - {6,8} can accept abb and a*b+.
  - Since the abb is listed first, it is the pattern of {6,8}.

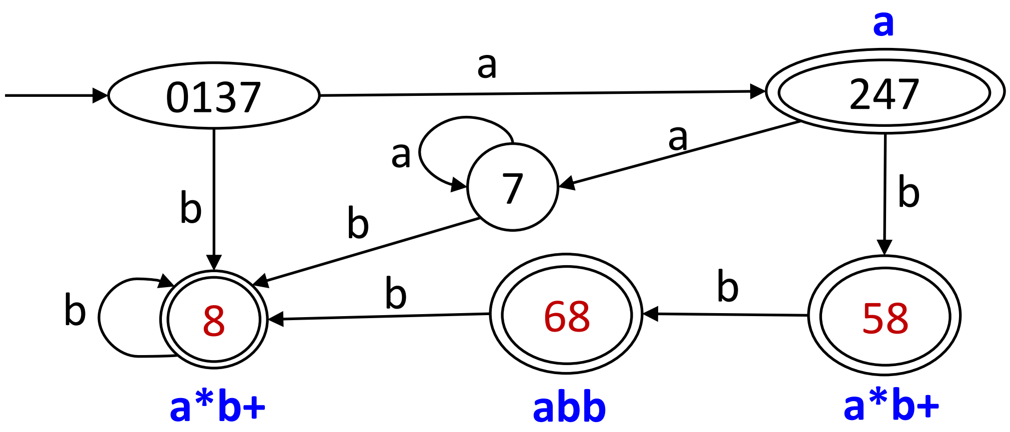
I	I_a	I_b
{0, 1, 3, 7}	{2, 4, 7}	{8}
{2, 4, 7} a	{7}	{5, 8}
{8} a*b+		{8}
{7}	{7}	{8}
{5, 8} a*b+		{6, 8}
{6, 8} abb, a*b+		{8}



Dragon book Fig. 3.54

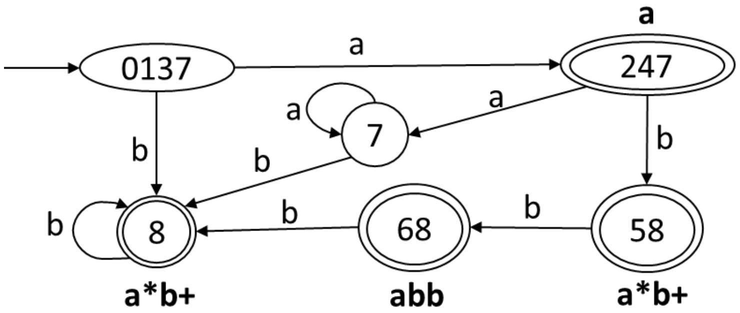
Lex: Example

- **Question**
  - Is this DFA minimal?
  - are 8, 68, 58 really equivalent?



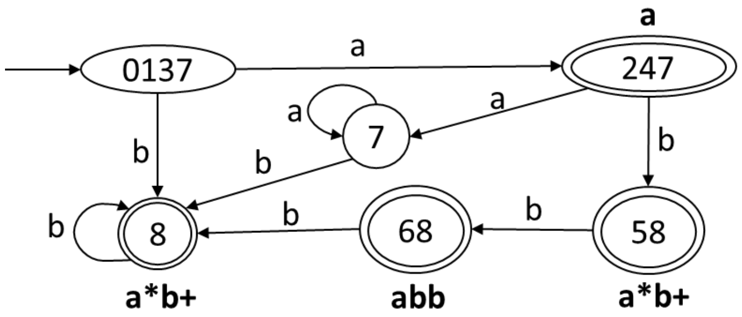
Lex: Example

- Depends on the language and the implementation approach
  - if abb is a keyword
    - **Approach 1: identify abb explicitly by FA with precedence.**
    - Approach 2: identify abb by an extra table
- Initial partition:
  - Non-accepting, accepting
  - {0137, 7}, {247}, {8, 58}, {68}
- Split {0137, 7}
  - move to different partitions on ‘a’
- Split {8, 58}
  - move to different partitions on ‘b’

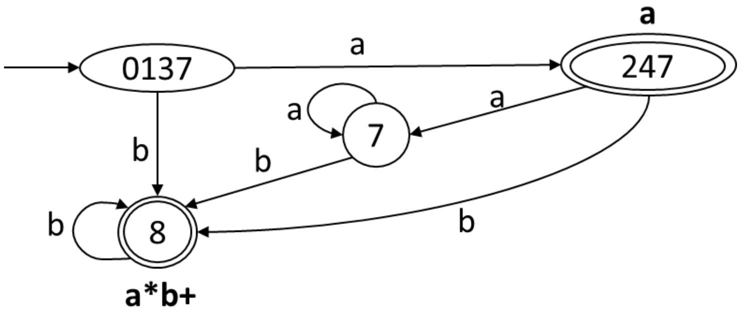


Lex: Example

- Depends on the language and the implementation approach
  - if abb is a keyword
    - Approach 1: identify abb explicitly by FA
    - **Approach 2: identify abb by an extra table**
  - Or it is just an identifier



- Initial partition:
  - non-accepting: {0137, 7},
  - accepting a: {247},
  - accepting a*b+: {58, 68, 8},
- Cannot split {58, 68, 8}
  - No move on 'a'
  - Move to {68, 8} on 'b'



The Limits of Regular Languages

- For ∑={a, b}
- The set of strings S over this alphabet consisting of a single b surrounded by **the same number** of a.

  S = {b, aba, aabaa, aaabaaa, …}

  L = $\{a^n b a^n$ | n ≥ 0}

the regular expression is?

a*ba* ✗

This set cannot be described by a regular expression

The Limits of Regular Languages

- L = $\{a^n b a^n \mid n \geq 0\}$ is not a Regular Language
  - FA does not have any memory (FA cannot count)
    - The above L requires to keep count of a's before seeing b's

- Matching parenthesis is not a RL[括号匹配不是正则语言]
- Any language with nested structure is not a RL

  if … if … else … else
- Regular Languages
  - Weakest formal languages that are widely used [最弱的形式语言]
- We need a more powerful formalism

Beyond Regular Language

- Regular languages are expressive enough for tokens
  - Can express identifiers, strings, comments, etc.
- However, it is the weakest (least expressive) language
  - Many languages are not regular
  - C programming language is not
    - The language matching braces "{{{...}}}" is also not
  - FA does not have any memory (FA cannot count)
    - L = $\{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures[嵌套结构] (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)

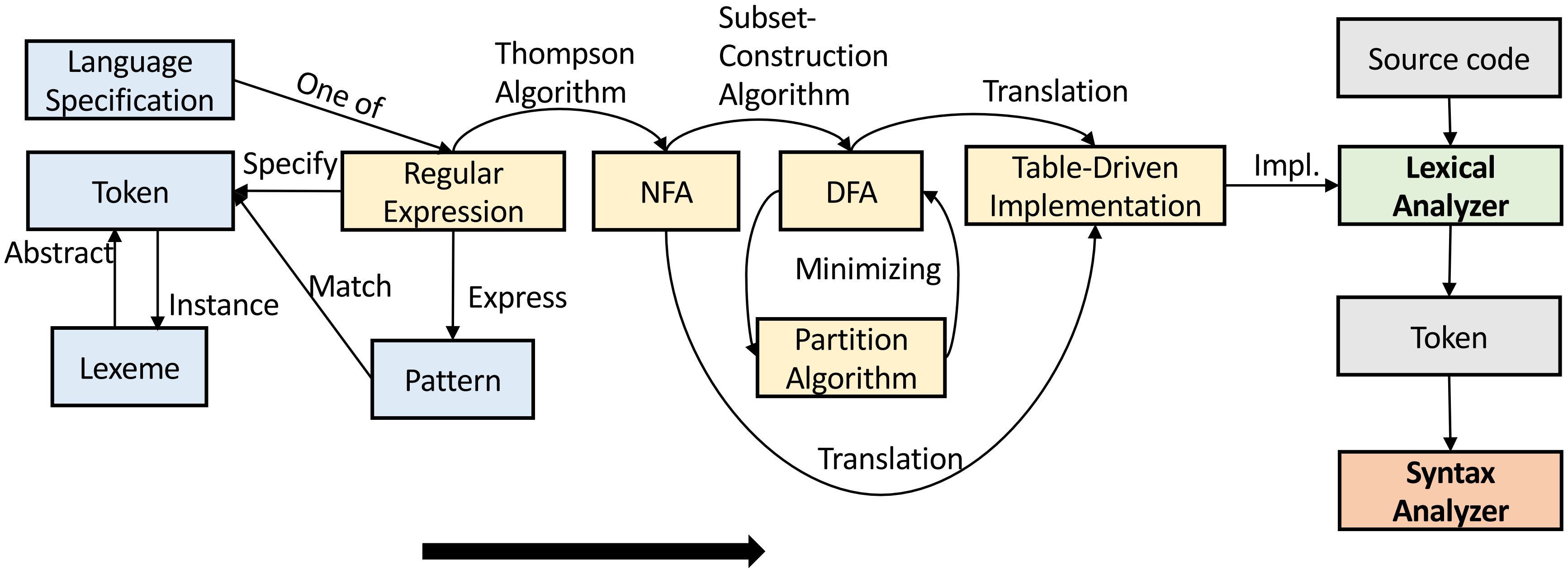
Summary

Lexical Specification → Regular expressions → NFA → DFA → Table-driven Implementation

Transition Flow

1. **Converting REs to NFA**
   - Thompson Algorithm(Inductive method)
2. **Converting NFA to DFA**
   - Subset-Construction Algorithm[子集构造法]
3. **Minimizing DFA**
   - Partition Algorithm[分割法]

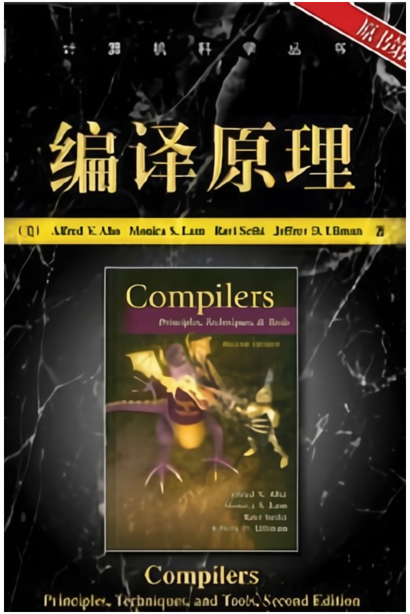
Summary



From Specification to Implementation

Further Reading

- Dragon Book
  - ◆ Comprehensive Reading:
    - ▫ Section Section 3.6–3.7, 3.9.6 for finite automata and related transformation.
  - ◆ Skip Reading:
    - ▫ Section 3.9.1–3.9.5 for regular expressions directly to DFAs.



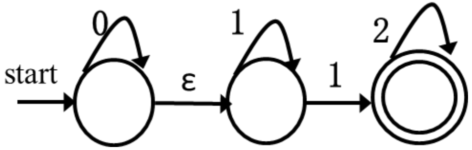
Exercise

- The graph describes NFA or DFA? Why?

  NFA.

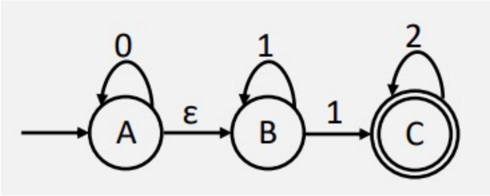
  A: ε-transition,

  B: multiple transitions for input ‘1’

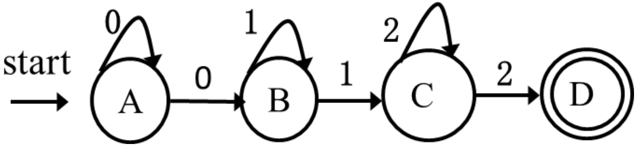


- What is the RE?

  0*1+2*



- Then, what is the NFA of 0+1+2+?



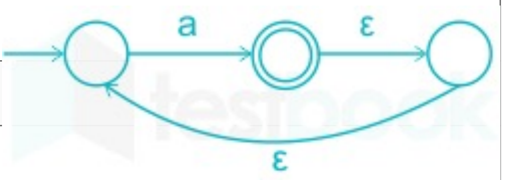
Exercise

SUN YAT-SEN UNIVERSITY

Que.	The behavior of a NFA can be simulated by a DFA
a.	always
b.	sometime
c.	Never
d.	Depend on NFA

Exercise

Que.	What is the complement[补集] of the language accepted by the NFA shown below? Assume ∑ = {a} and ε is the empty string
a.	Φ
b.	ε
c.	a
d.	{ε, a}



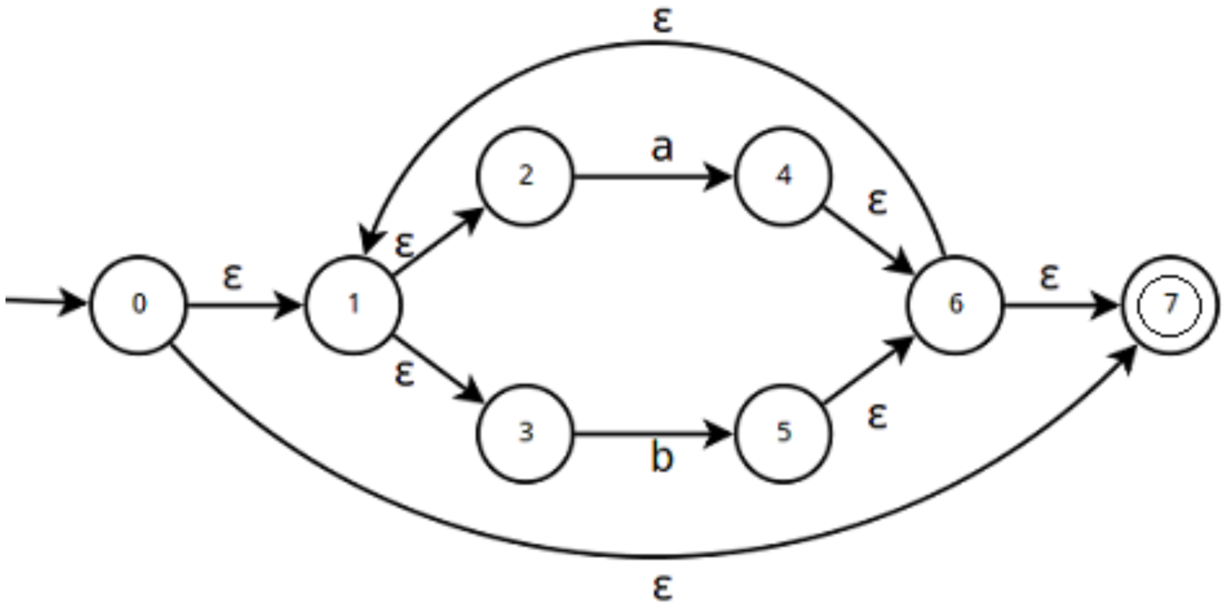
Exercise

- Convert (a|b)*abb(a|b)* into NFA
  - Thompson construction: RE→NFA

    (a|b)→(a|b)*

    abb

    (a|b)*abb(a|b)*



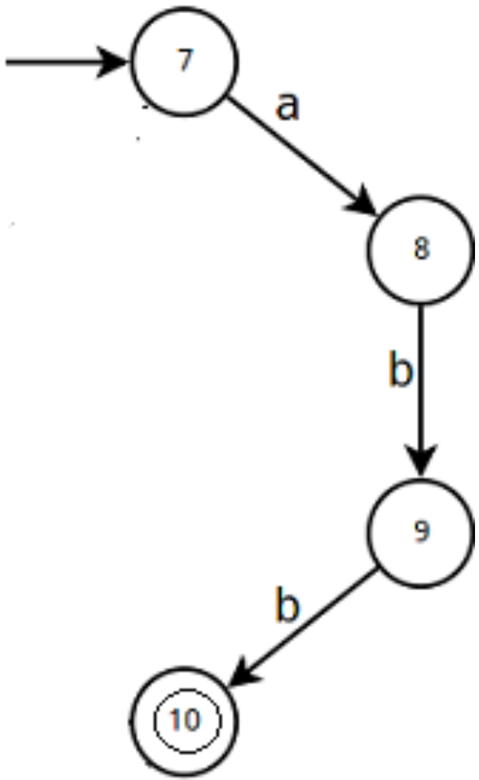
Exercise

- Convert (a|b)*abb(a|b)* into NFA
  - Thompson construction: RE→NFA

    (a|b)→(a|b)*

    abb

    (a|b)*abb(a|b)*



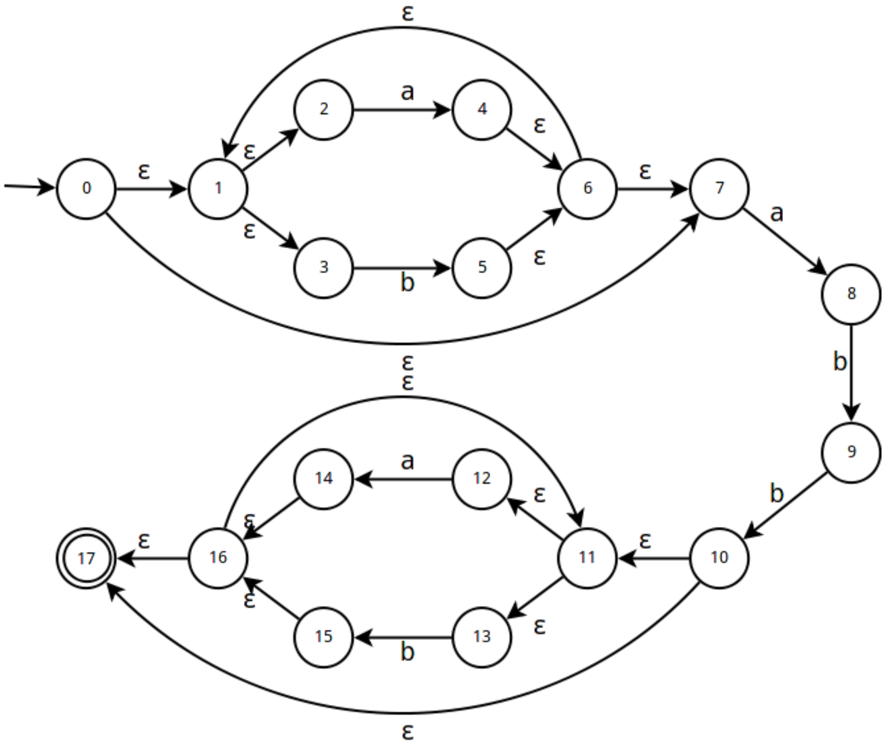
Exercise

- Convert (a|b)*abb(a|b)* into NFA
  - Thompson construction: RE→NFA

    (a|b)→(a|b)*

    abb

    (a|b)*abb(a|b)*



Exercise: from RE to minimized FA

- Construct a DFA for a minion language with $\sum = \{a, b\}$ that does not contain "abb":

1. Build the regular expression for the minion's language
2. Convert the regular expression into NFA first
3. Convert the NFA into DFA by subset construction.
4. Minimize the state of DFA

https://www.sensacine.com/peliculas/pelicula-210493/fotos/detalle/?cmediafile=21209746

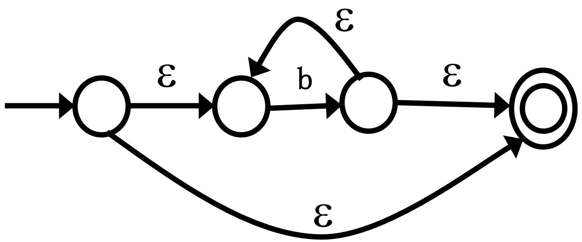
- Build the RE for this language

  b*(a | ab)*

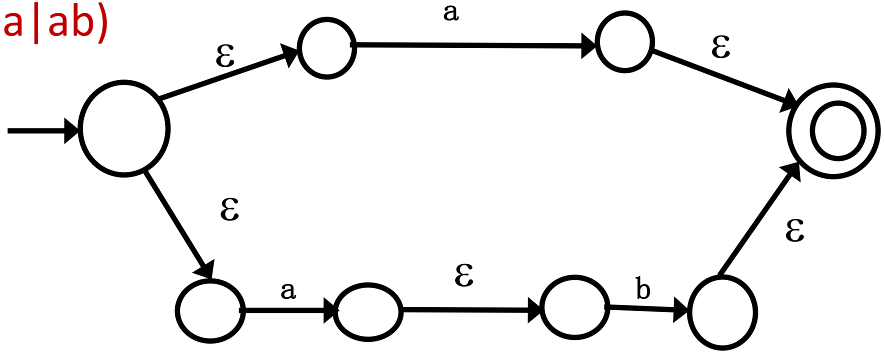
Exercise

- Convert b*(a | ab)* into NFA

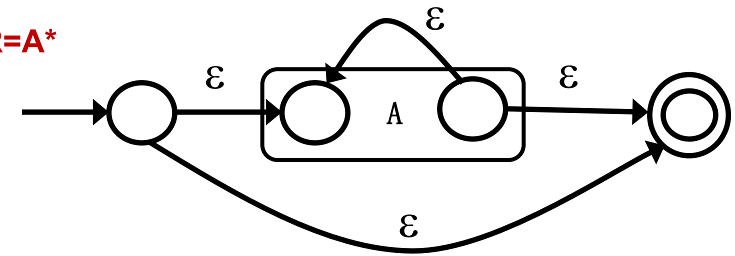
b*



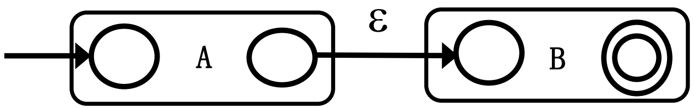
(a|ab)



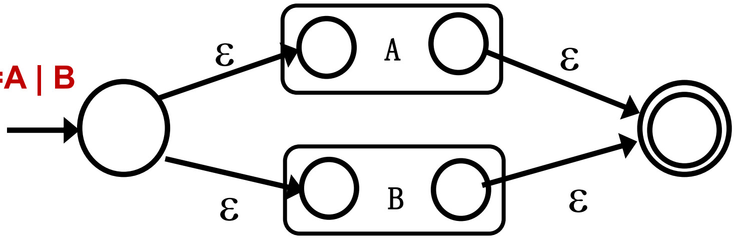
R=A*



R=AB

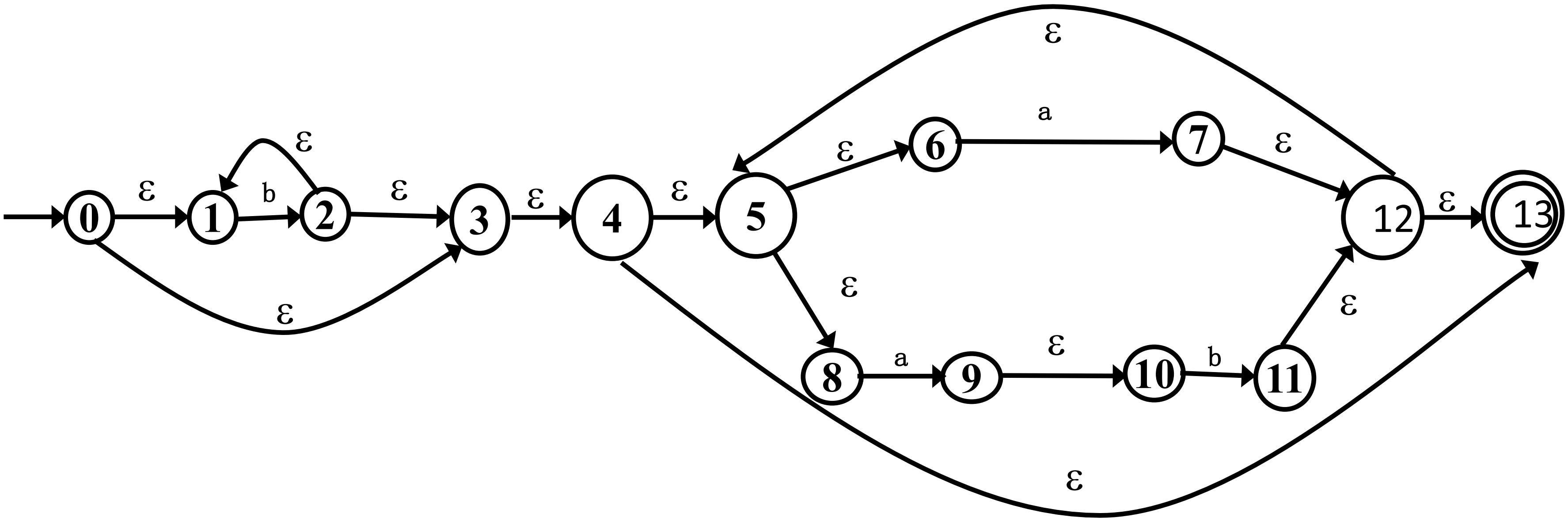


R=A | B

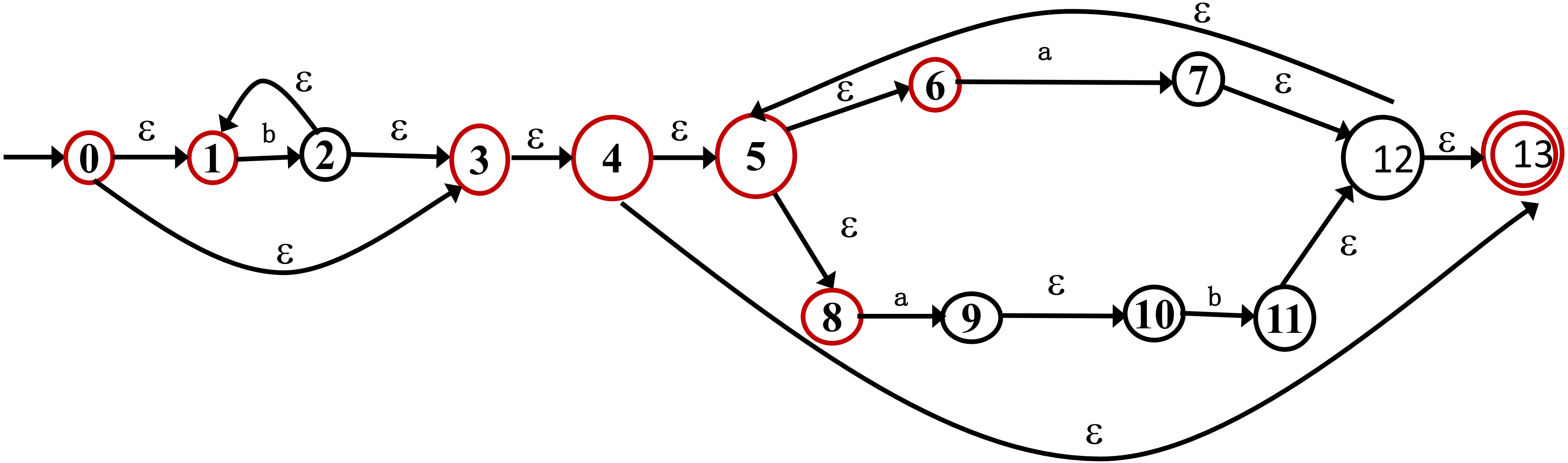


Exercise

- Convert the NFA into DFA by subset construction



Exercise

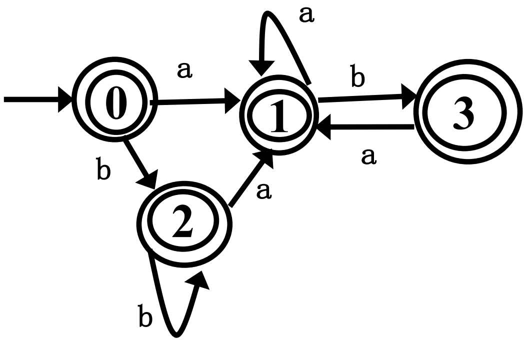


I	I_a	I_b	Accept
{0,1,3,4,5,6,8,13} 0	{7,9,12,13,5,6,8,10} 1	{2,1,3,4,5,6,8,13} 2	Yes
{5,6,7,8,9,10,12,13} 1	{5,6,7,8,9,10,12,13} 1	{11,12,13,5,6,8} 3	Yes
{1,2,3,4,5,6,8,13} 2	{5,6,7,8,9,10,12,13} 1	{1,2,3,4,5,6,8,13} 2	Yes
{5,6,8,11,12,13} 3	{5,6,7,8,9,10,12,13} 1	{}	Yes

Exercise

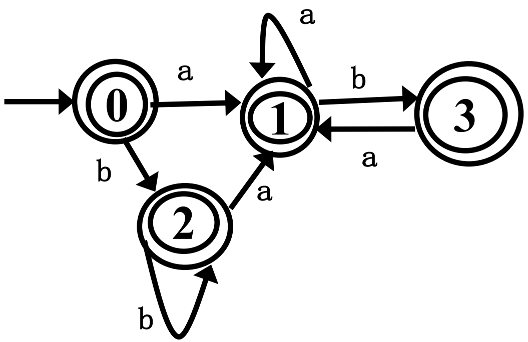
- Draw DFA according to the transition table

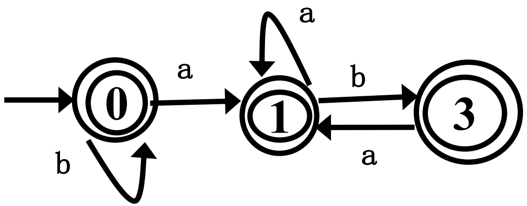
I	I_a	I_b
{0,1,3,4,5,6,8,13} 0	{7,9,12,13,5,6,8,10} 1	{2,1,3,4,5,6,8,13} 2
{5,6,7,8,9,10,12,13} 1	{5,6,7,8,9,10,12,13} 1	{11,12,13,5,6,8} 3
{1,2,3,4,5,6,8,13} 2	{5,6,7,8,9,10,12,13} 1	{1,2,3,4,5,6,8,13} 2
{5,6,8,11,12,13} 3	{5,6,7,8,9,10,12,13} 1	{}



Exercise

- Minimization DFA





- Initial: {0,1,2,3}
  {3} have no 'b' transition， split
- {0,1,2} {3}
  {1} on 'b' ->3， {0,2} on 'b' ->{2}， split
- {0,2} {1} {3}
  {0,2} on 'a' -> {1}
  {0,2} on 'b' -> {2}
  No way to distinguish {0,2} on any transition with 'a' or 'b'
- Final: {0,2} {1} {3}