编译原理
Complier Principles

Lecture2
Lexical Analysis: NFA&DFA

赵帅

计算机学院

中山大学

Finite Automata[有穷自动机]

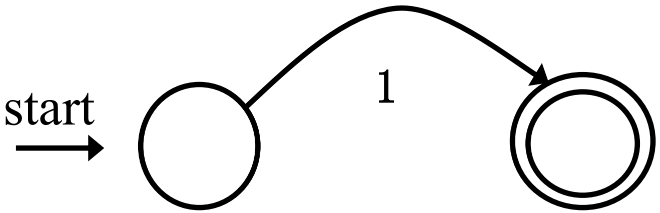
- **REs is only a language specification**[只是定义了语言]
  - to construct a token recognizer for languages given by regular expressions
- **How do we go from specification to implementation?**
  - Regular expressions can be implemented using finite automata
  - There are two types of automata
    - **NFAs (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

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

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

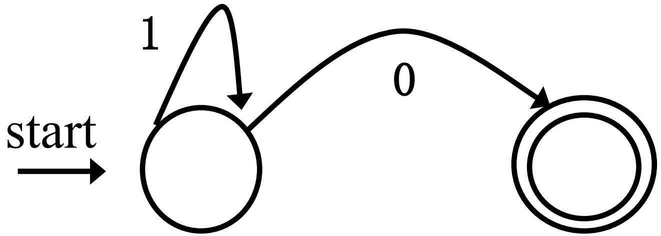
- Finite Automata are similar to transition diagrams
  - They have states and labelled edges
  - There are one unique start state and one or more than one final states

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? $\sum = \{0, 1\}$
  - Any number of '1's followed by a single 0

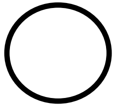
DFA and NFA

- 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

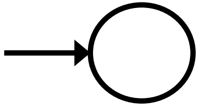
- Nondeterministic Finite Automata (NFA): the machine can exist in multiple states at the same time[非确定的有限状态机]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at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

State Graph

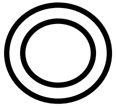
- 5 components (∑, S, n, F, δ)
  - An input alphabet Σ
  - A set of states S



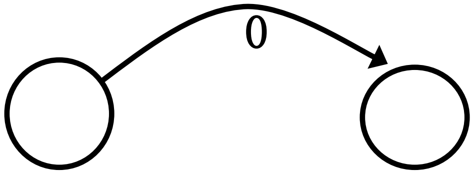
  - A start state $n \in S$



  - A set of accepting states $F \subseteq S$

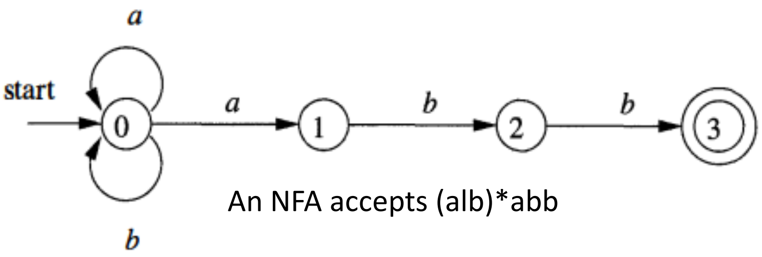


  - A set of transitions δ: $S_a \xrightarrow{\text{Input}} S_b$



Comparison of NFA and DFA

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

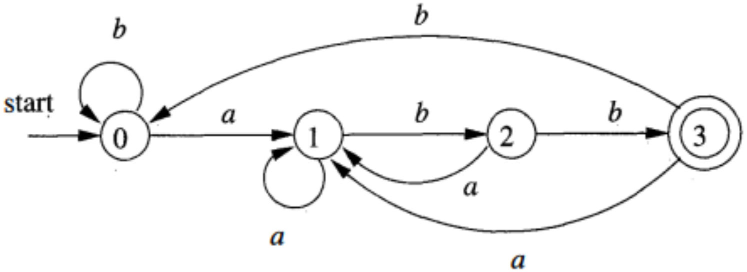


An NFA accepts (a|b)*abb

  - 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

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

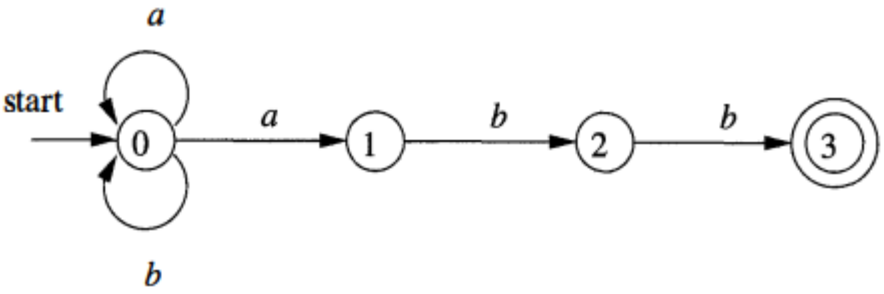


A DFA accepts (alb)*abb

  - 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**

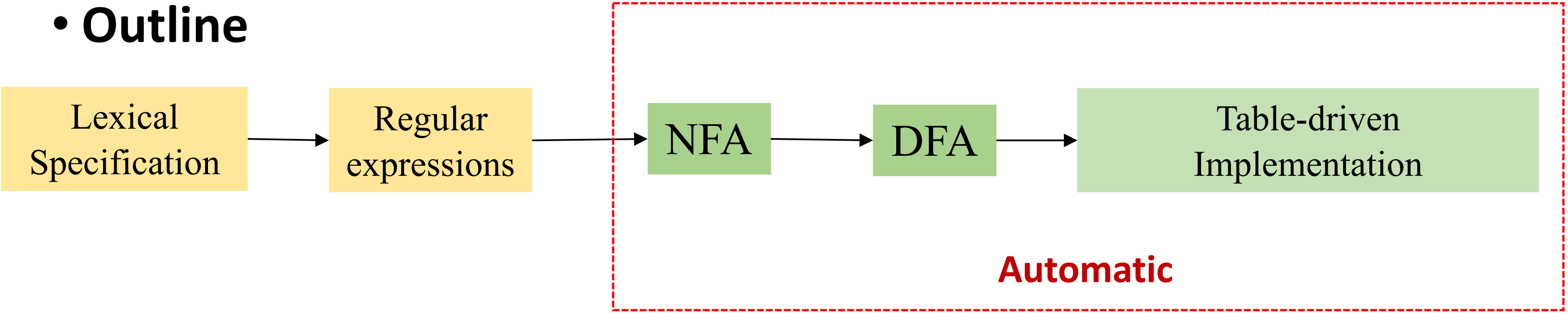


State	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

- 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.

Conversion Flow[转换流程]

- **Outline**



① **Converting REs to NFAs**

② **Converting NFAs to DFAs**

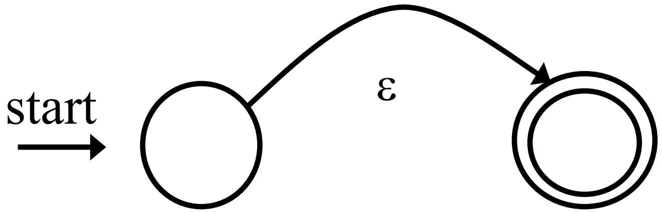
③ **Converting DFAs to table-driven implementations**

Construct NFA for RE

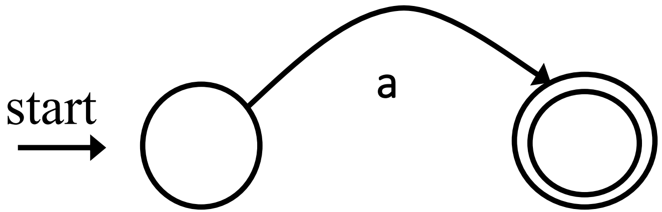
（Thompson算法）

Basic: processing atomic REs

- NFA for ε



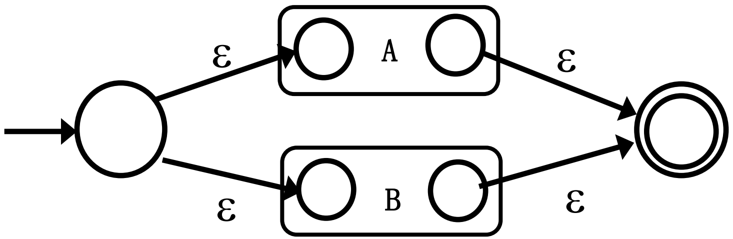
- NFA for single character a



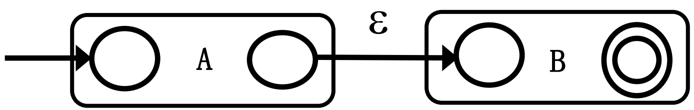
Construct NFA for RE

Inductive: processing compound Res

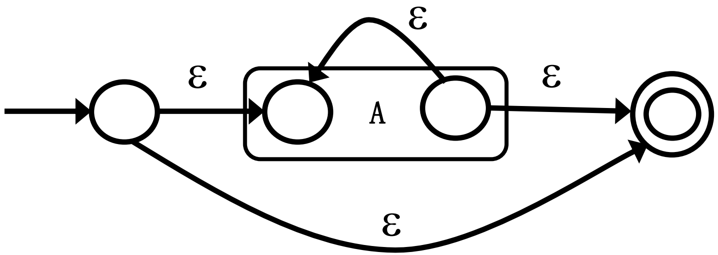
R=A|B



R=AB



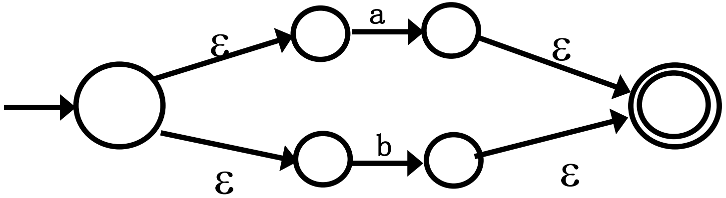
R=A*

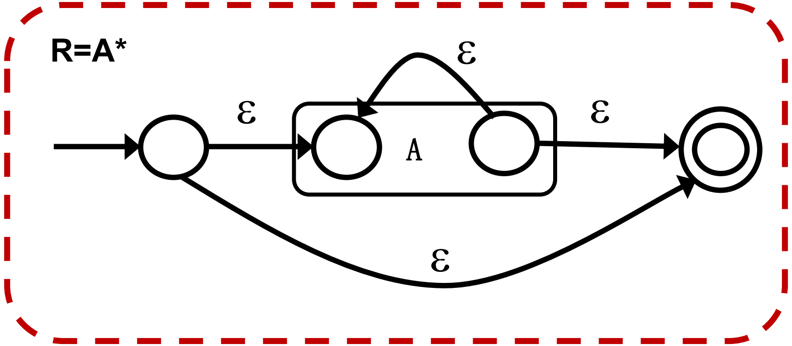


Example

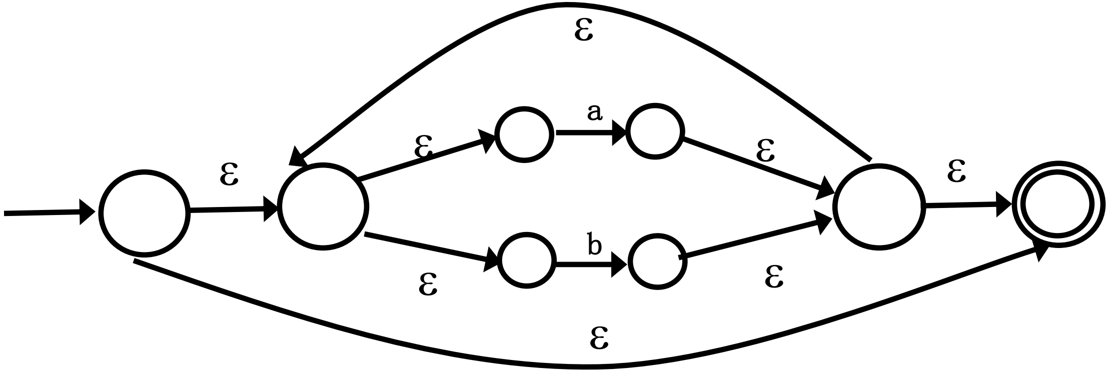
- Convert "(a|b)*abb" to NFA

(a|b)





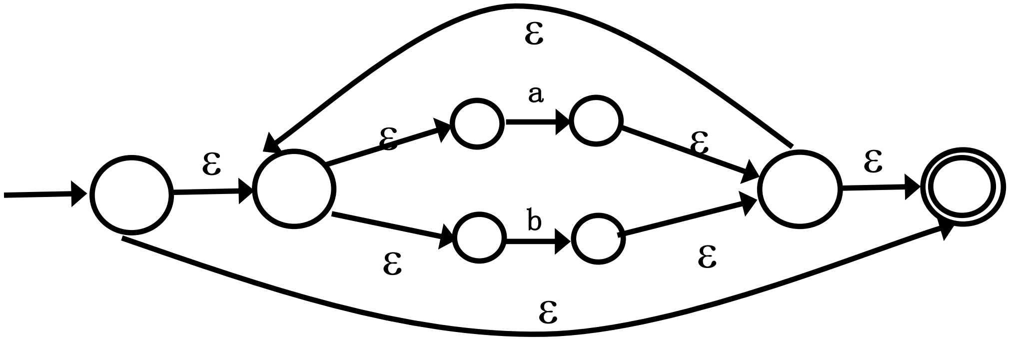
(a|b)*



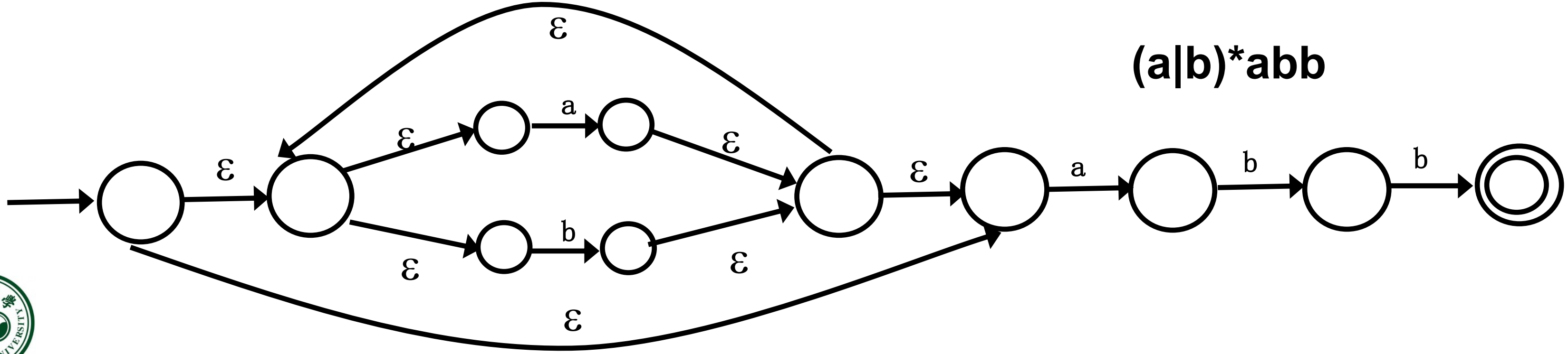
Example

- Convert “(a|b)*abb” to NFA

(a|b)*

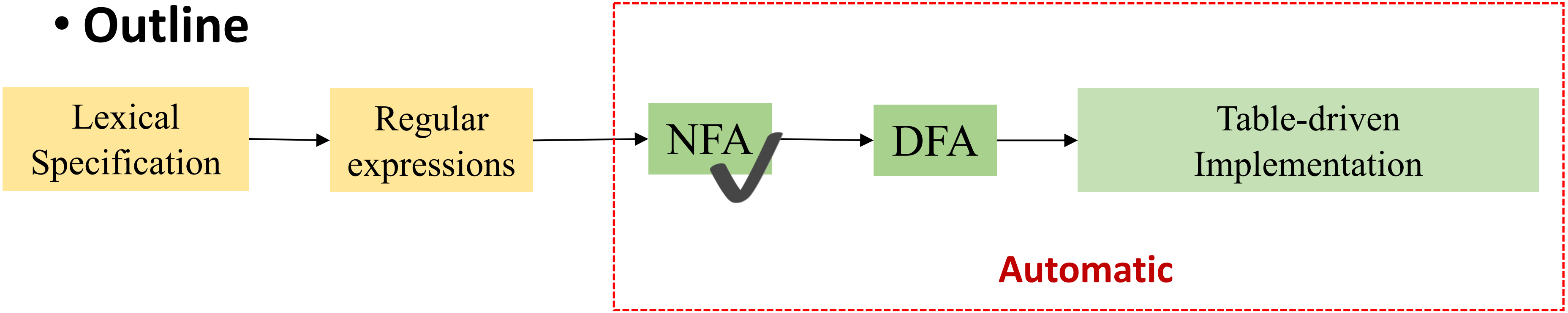


(a|b)*abb



Conversion Flow[转换流程]

- **Outline**



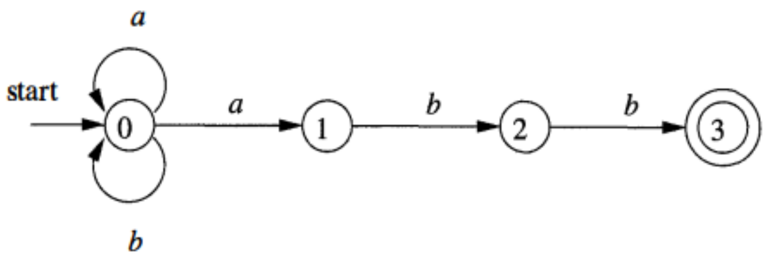
① **Converting REs to NFAs**

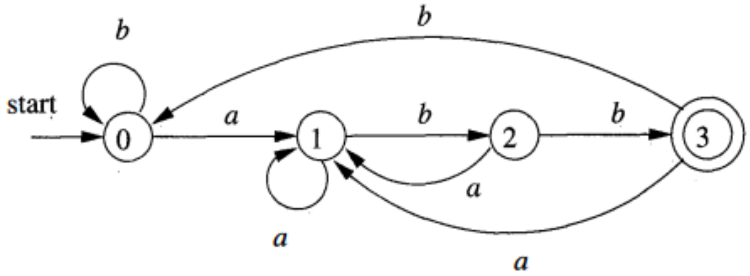
② **Converting NFAs to DFAs**

③ **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

中山大學 SUN YAT-SEN UNIVERSITY

- Theorem: L(NFA) ≡ L(DFA)
  - Both recognize regular languages L(RE)
  - Will show L(NFA) ⊆ L(DFA) by construction (NFA→ DFA)
- 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

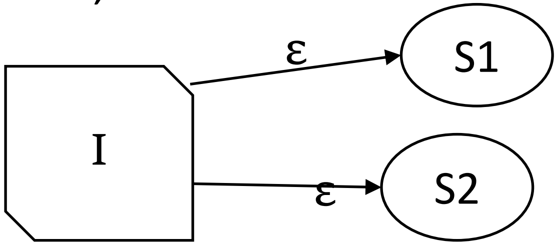
- Recall DFA
  - Every state must have exactly one transition defined for every letter
  - ε-moves are not allowed
    - NFAs have multiple transition, while DFAs can only have one transition in one time

- **Subset construction**[子集构造法]
  - **Each state of the constructed DFA corresponds to a set of NFA states**
    - After reading input $a_1 a_2 \dots a_n$, 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 $a_1 a_2 \dots a_n$,

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**



$\varepsilon\text{-closure}(I) = I \cup \{S1, S2\}$

From NFA to DFA: Algorithm

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

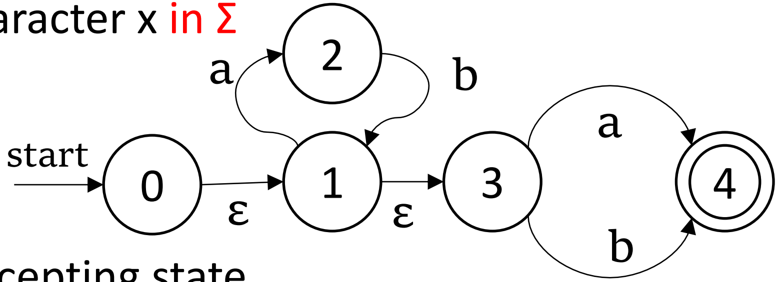
I is a set of states, a is a character in the alphabet

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

I_a = ε-closure($move(I,a)$)

Example

- Step1: Start by constructing ε-closure of the start state
  - I = ε-closure(s0) = {0, 1, 3}
- Step2: Keep getting ε-closure(move(I, x)) for each character x in Σ
  - I_a= ε-closure(move(I,a))={2,4}
  - I_b = ε-closure(move(I,b)) = {4}
- Stop, when there are no more new states
- Mark as accepting for those states that contain an accepting state

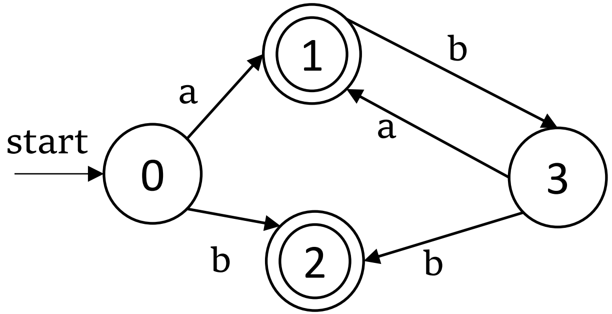


I	I_a	I_b	Accept
{0, 1, 3} mark **T0**	{2, 4} mark T1	{4} mark T2	**T0** No
{2, 4} **T1**		{1, 3} mark T3	**T1** Yes
{4} **T2**			**T2** Yes
{1,3} **T3**	{2,4} T1	{4} T2	**T3** No

Example

- **Construct DFA**

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



- **Is the DFA minimal?**

Minimizing DFA

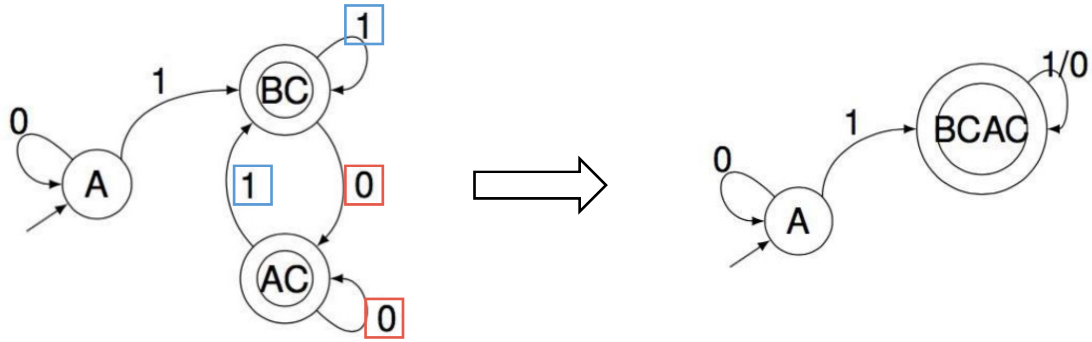
- **Theory:** Given any DFA, there is an equivalent DFA containing a minimum number of states, and this minimum-state DFA is unique

- **Equivalent States**

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

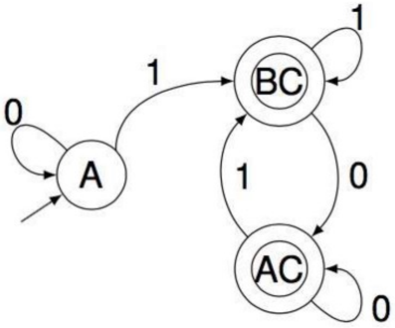
  ① s and t are both accepting states or both non-accepting states.

  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

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**

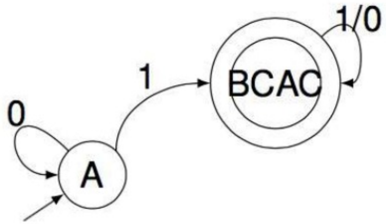
  For {BC, AC}

  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}



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 nonaccepting 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

Example: Minimization

① **Initial**

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

② **Check the transitions**

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

Move(I_1 , a) = {C, F} and {C, F} is the subset of {C,D,E,F}.

Move(I_1 , b) = {D, E} and {D, E} is the subset of {C,D,E,F}.

{C, D, E, F} are equivalent

For I_2 = {S, A, B}

Move({S, B}, a) = {A}, Move({A}, a) = {C} no eq

So splitting {S, A, B} →{S, B}, {A}

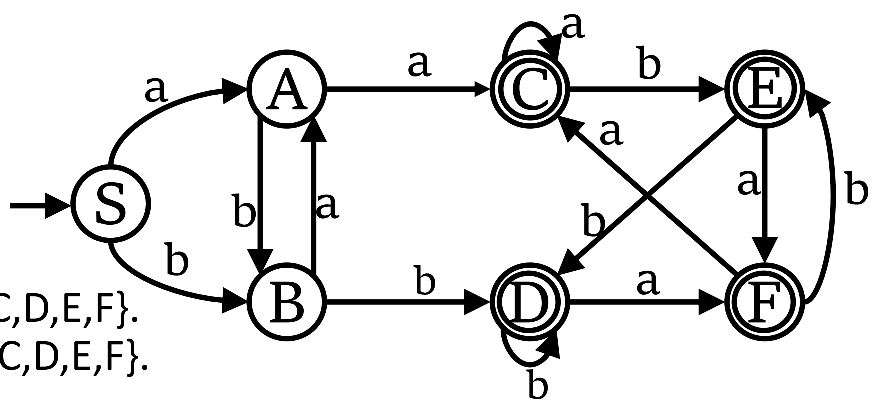
Check {S, B}, Move({S}, b) = {B}, Move({B}, b) = {D}

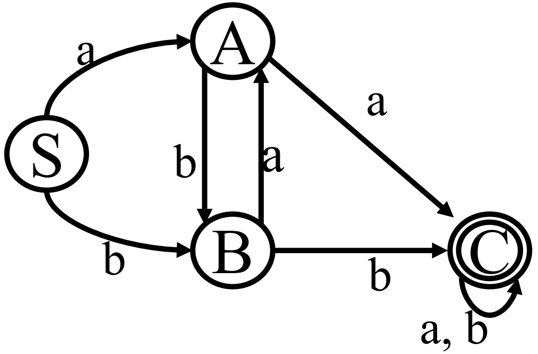
So splitting {S, B} →{S}, {B}

③ **Finally, get the subsets and draw min DFA**

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

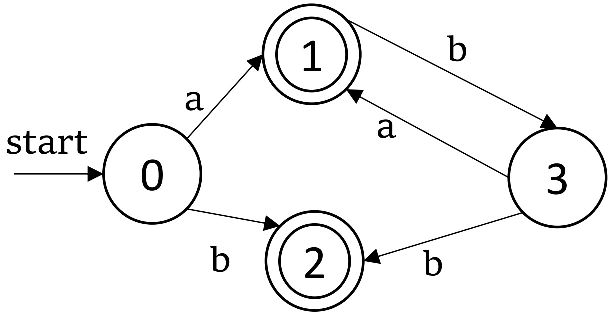
{C, D, E, F} denotes {C}





Example

- **Is the DFA minimal?**

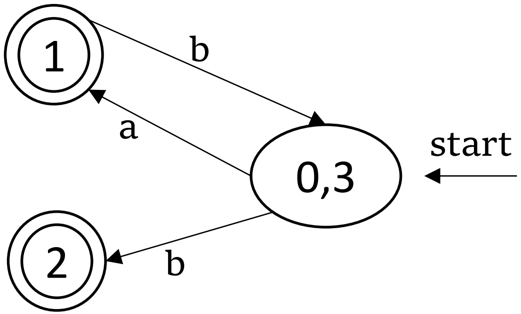


Result: {0, 3} {1} {2}

For I_1 = {0,3}

Move(I_1 , a) = {1}

Move(I_1 , b) = {2}.



0 and 3 are equivalent states

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

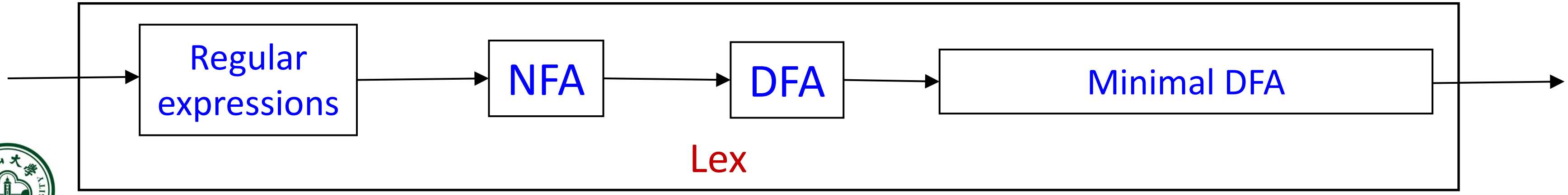
NFA → DFA: Time Complexity[复杂度]

中山大學 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|)
- 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)

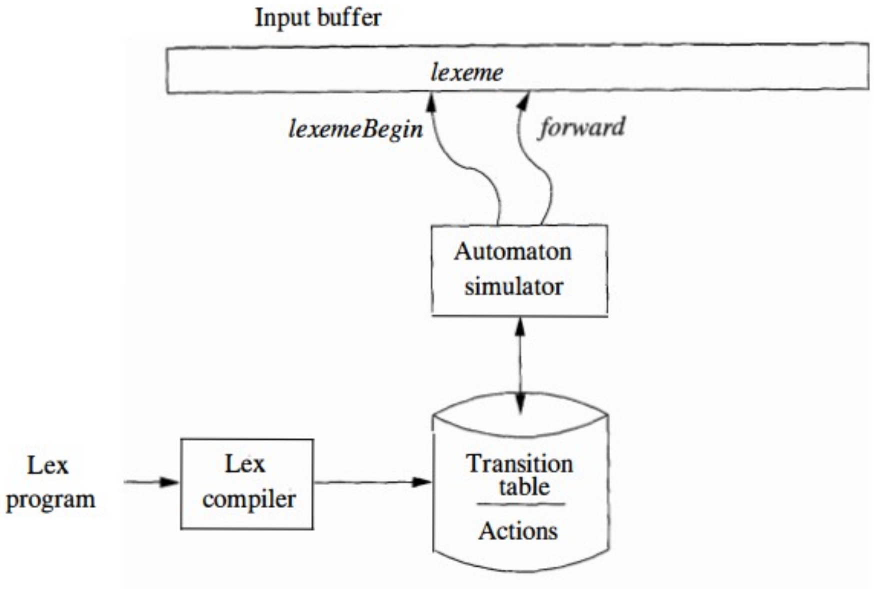
Implementation in Practice[实际实现]

- **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

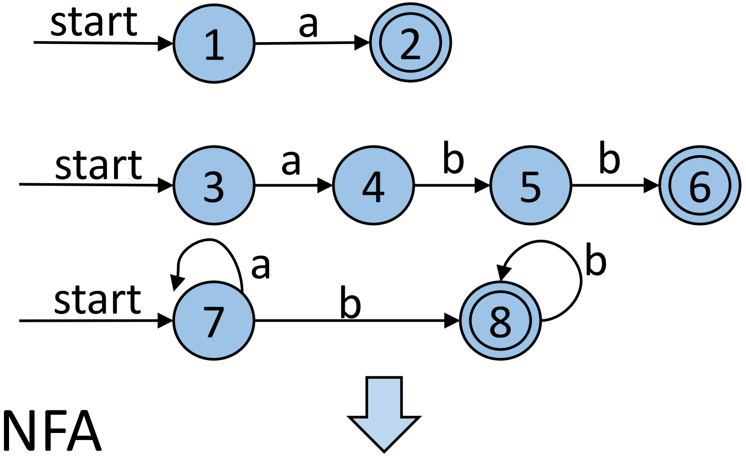
- A Lex program is turned into a transition table and actions, which are used by a FA simulator
- Automaton need to recognize lexemes matching any of the patterns in a program



Lex: Example

- Three patterns, three NFAs

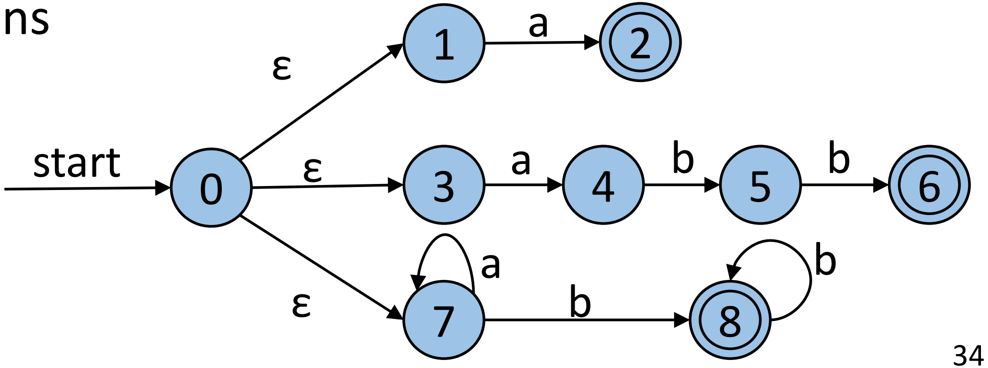
a	{action$_1$}
abb	{action$_2$}
a*b+	{action$_3$}



- Combine three NFAs into a single NFA

  Add start state 0 and ε-transitions

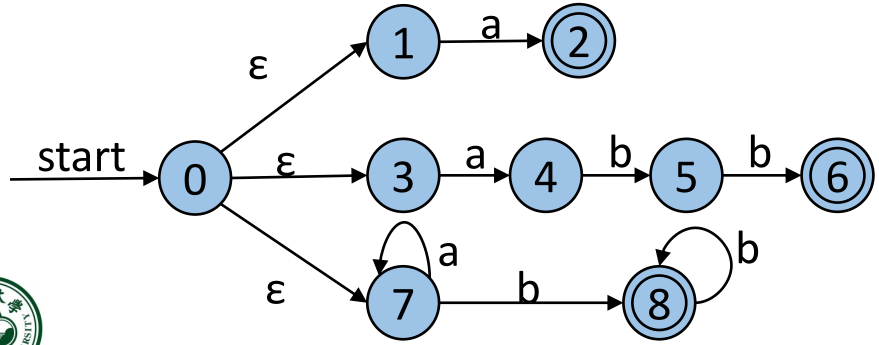
Any one is possible,
if you haven't read
any input symbol

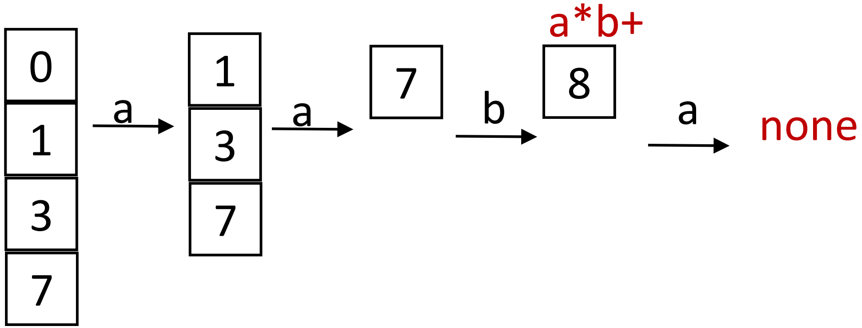


Lex: Example

- 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

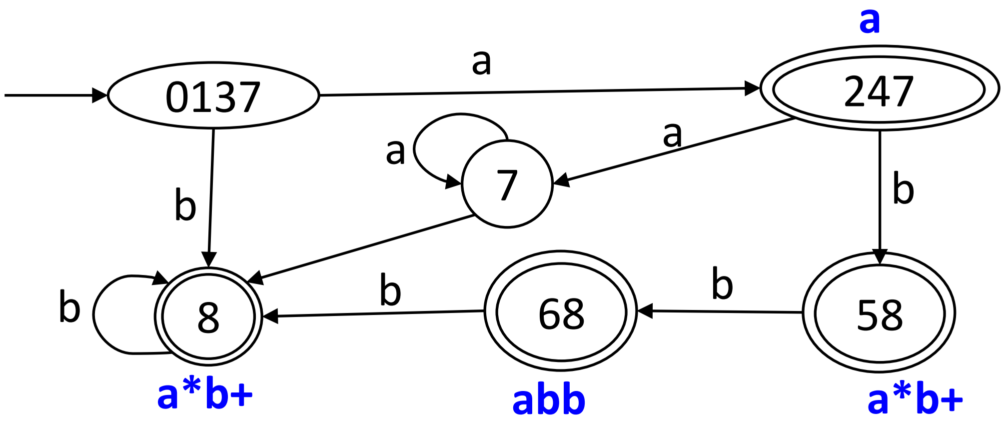
    Back up, looking for a set of states that include an accepting state
  - State 8: a*b+ has been matched
  - Select aab as the lexeme, execute action3
    - Return to parser indicating that token with pattern a*b+ has been found





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 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, but you can implement approach 1 in your projects for simplicity

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]

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.