中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle
# 编 译 原 理

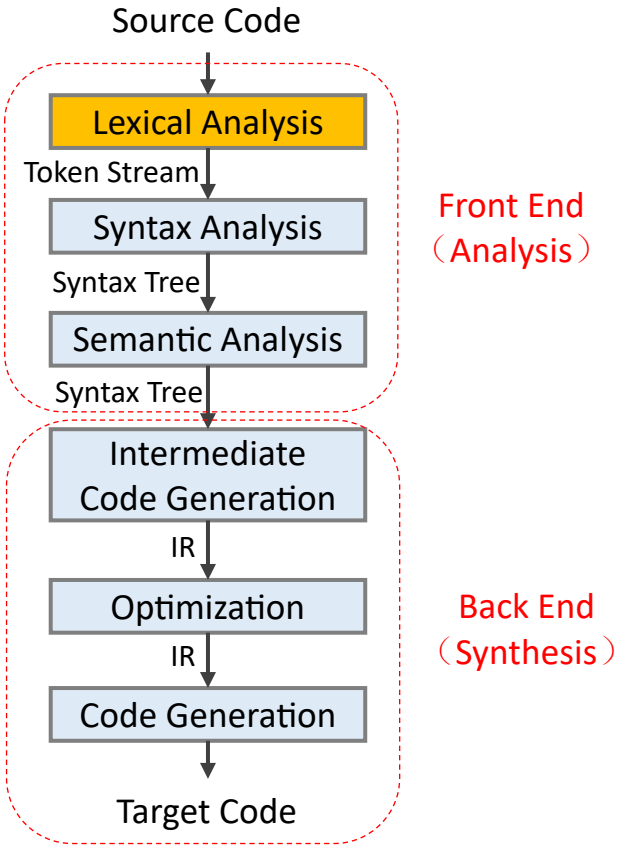
## 第1讲：词法分析(1)

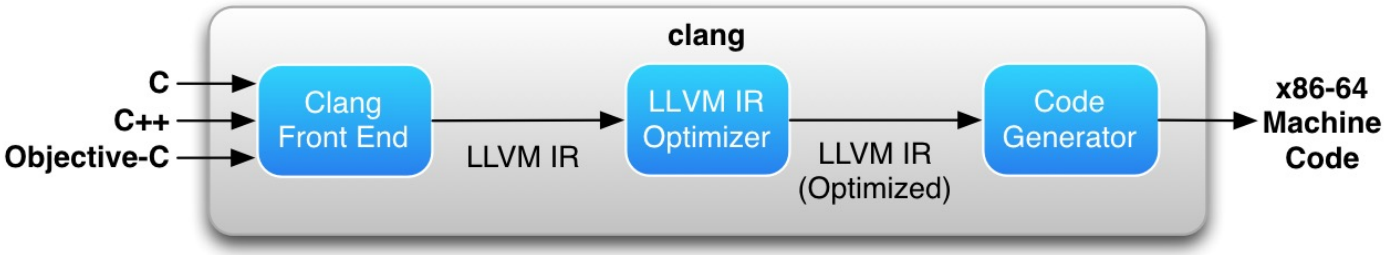
张献伟

xianweiz.github.io

DCS290, 2/29/2024

# Structure of a Typical Compiler[结构]



# What is Lexical Analysis[词法分析]?

- Example:

```
/* simple example */
if (i == j)
  z = 0;
else
  z = 1;
```

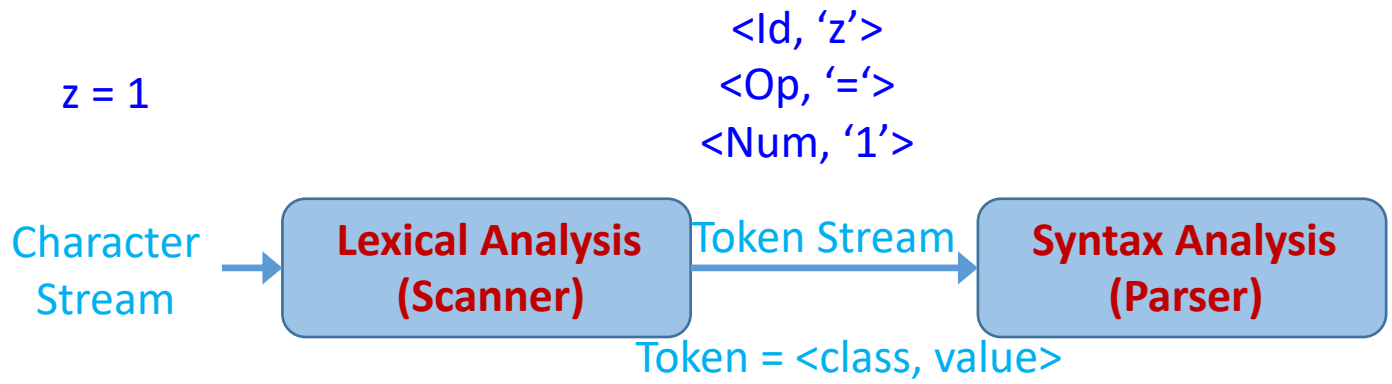
- Input[输入]: a string of characters
  - "*if* (*i* == *j* )\n\t*z* = 0; \n*else*\n\t*z* = 1; \n"
- Goal[目标]: partition the string into a set of substrings
  - Those substrings are **tokens**
- Steps[步骤]
  - Remove comments: ~~/* simple example */~~
  - Identify substrings: 'if' '(' 'i' '==' 'j' ……
  - Identify **token classes**: (keyword, 'if'), (LPAR, '('), (id, 'i') ……

# What is a token[词]?

- **Token**: a “word” in language (smallest unit with meaning)
  - Categorized into classes according to its role in language
  - Token classes in English[自然语言]
    - Noun, verb, adjective, …
  - Token classes in a programming language[编程语言]
    - Number, keyword, whitespace, identifier, …
- Each **token class** corresponds to a set of strings[类: 集合]
  - Number: a non-empty string of digits
  - Keyword: a fixed set of reserved words (“for”, “if”, “else”, …)
  - Whitespace: a non-empty sequence of blanks, tabs, newlines
  - Identifier: user-defined name of an entity to identify
    - Q: what are the rules in C language?

# Lexical Analysis: Tokenization[分词]

- Lexical analysis is also called **Tokenization** (also called Scanner)[扫描器]
  - Partition input string into a sequence of tokens
  - Classify each token according to its role (token class)
    - **Lexeme**[词素]: an instance of the token class, e.g. ‘z’, ‘=’, ‘1’
- Pass tokens to syntax analyzer (also called Parser)[分析器]
  - Parser relies on token classes to identify roles (e.g., a *keyword* is treated differently from an *identifier*)



# Lexical Analyzer: Design[设计]

- Define a finite set of token classes[定义token类别]
  - Describe all items of interest
  - Depends on language, design of parser
  - "*if* (*i* == *j*)\n\t*z* = 0; \n*else*\n\t*z* = 1; \n"
    - Keyword, identifier, whitespace, integer

- Label which string belongs to which token class[识别]

```
if (i == j)
  z = 0;
else
  z = 1;
```

'==' or '='?

keyword or identifier?

# Lexical Analyzer: Implementation[实现]

- An implementation must do two things
  - Recognize the token class the substring belongs to[识别分类]
  - Return the value or lexeme of the token[返回对应值]
- A token is a tuple (class, lexeme)[二元组]

- The lexer usually discards "non-interesting" tokens that don't contribute to parsing[丢弃无意义词]
  - e.g., whitespace, comments
- If token classes are non-ambiguous, tokens can be recognized in a single left-to-right scan of input string
- Problem can occur when classes are ambiguous[歧义]

# Ambiguous Tokens in C++

- C++ template syntax
  - Foo<Bar>
- C++ stream syntax
  - cin >> var

Template: a blueprint or formula for creating a generic class or a function.
Templates are expanded at compiler time, similar to macros.

```
Template <typename T>
T getMax(T x, T y) {
   return (x > y) ? x : y;
}

int main (int argc, char* argv[]) {
    getMax<int>(3, 7);
    getMax<double>(3.0, 2.0);
    getMax<char>('g', 'e');

    return 0;
}
```

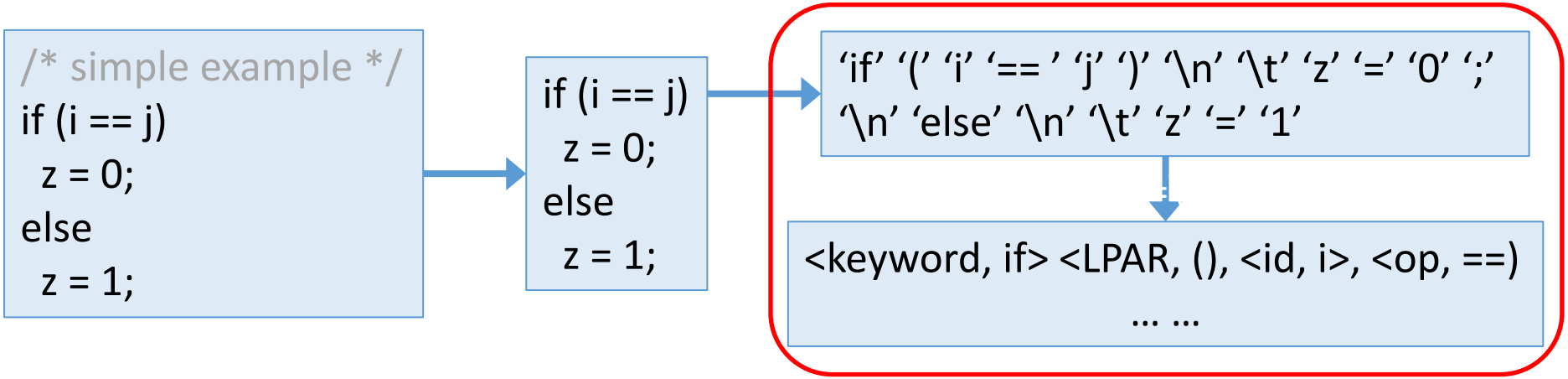
- Ambiguity
  - Foo<Bar<Bar>>
  - cin >> var
  - Q: Is '>>' a stream operator or two consecutive brackets?

# Look Ahead[展望]

- "look ahead" may be required to resolve ambiguity[展望消除歧义]
  - Extracting some tokens requires looking at the larger context or structure[需要上下文或语法结构]
  - Structure emerges only at parsing stage with parse tree[后一阶段才有]
  - Hence, sometimes feedback from parser needed for lexing
    - This complicates the design of lexical analysis
    - Should minimize the amount of look ahead
- Usually tokens do not overlap[通常无重叠]
  - Tokenizing can be done in one pass w/o parser feedback
  - Clean division between lexical and syntax analyses

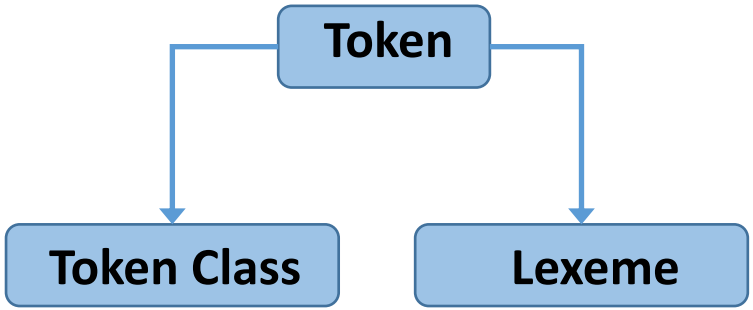
# Summary: Lexer

- Lexical analysis
  - Partition the input string to lexeme
  - Identify the token class of each lexeme
- Left-to-right scan => look ahead may be required
  - In reality, lookahead is always needed
  - The amount of lookahead should be minimized



# Token Specification[定义]

- Recognizing token class: how to describe string patterns
  - i.e., which set of strings belong to which token class?
  - Use regular expressions[正则表达式] to define token class
- **Regular Expression** is a good way to specify tokens
  - Simple yet powerful (able to express patterns)
  - Tokenizer implementation can be generated automatically from specification (using a translation tool)
  - Resulting implementation is provably efficient



# Language: Definition

- **Alphabet** ∑[字母表]: a finite set of symbols
  - Symbol: *letter*, *digit*, *punctuation*, …
  - Example: {0, 1}, {a, b, c}, ASCII
- **String**[串]: a finite sequence of symbols drawn from ∑
  - i.e., sentence or word
  - Example: *aab* (length = 3), *ε* (empty string, length = 0)
- **Language**[语言]: a set of strings of the characters drawn from ∑
  - ∑ = {0, 1}, then {}, {01, 10}, {1, 11, 1111, …} are all languages over ∑
  - {ε} is a language
  - Φ, empty set is also a language

# Language: Example

- Examples:
  - Alphabet ∑ = (set of) English characters
    - Language *L* = (set of) English sentences
  - Alphabet ∑ = (set of) Digits, +, -
    - Language *L* = (set of) Integer numbers

- Languages are subsets of all possible strings
  - Not all strings of English characters are (valid) sentences
    - aaa bbb ccc
  - Not all sequences of digits and signs are integers
    - 125+, 1-25

# Language: Operations[语言运算]

- In lexical analysis, the most important operations on languages are union, concatenation and closure
- **Union**[并]: similar operation on sets
- **Concatenation**[连接]: all strings formed by taking a string from the first language and a string from the second language in all possible ways, and concatenating them
- **Kleene closure**[闭包]: $L^*$, where $L$ is the language, is the set of strings you get by concatenating $L$ zero or more times
  - $L^0 = \{\varepsilon\}$, $L^i = L^{i-1}L$
  - $L^+$: the same as Kleene closure, but without $L^0$
    - $L \cup L^2 \cup L^3 \cup \ldots$
    - ε won't be in $L^+$ unless it is in $L$ itself

# Example

- $\sum_1 = \{0, 1\}$, $\sum_2 = \{a, b\}$ ==> $L_1 = \{0, 1\}$, $L_2 = \{a, b\}$
- $L_1 \cup L_2$
  $= \{0, 1\} \cup \{a, b\} = \{0, 1, a, b\}$
- $L_1 L_2$
  $= \{0, 1\}\{a, b\} = \{0a, 0b, 1a, 1b\}$
- $L_1^3$
  $= \{0, 1\}^3 = \{0, 1\}\{0, 1\}\{0, 1\} = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- $L_1^*$
  $= L_1^0 \cup L_1^1 \cup L_1^2 \cup L_1^3 \cup \ldots = \{\varepsilon\} \cup \{0, 1\} \cup \{0, 1\}^2 \cup \{0, 1\}^3$
  $= \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$
- $L_1^+$
  – $= L_1^* - L_1^0 = \{0, 1\} \cup \{0, 1\}^2 \cup \{0, 1\}^3$
  – $= \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$

# Language: Example (cont.)

- $L$ = {A, B, …, Z, a, b, …, z}, D = {0, 1, …, 9}
  - $L$ and $D$ are also languages whose strings happen to be of length one
  - Some other languages that can be constructed from $L$ and $D$ are
- $L \cup D$: the set of letters and digits, i.e., language with 62 strings of length one
- $LD$: the set of 520 strings of length two, each is one letter followed by one digit
- $L^4$: the set of all 4-letter strings
- $L^*$: the set of all strings of letters, including ε, the empty string
- $L(L \cup D)^*$: the set of all strings of letters and digits beginning with a letter
- $D^+$: the set of all strings of one or more digits

**Identifiers can be described by giving names to sets of letters and digits and using the language operators**

# Regular Expressions & Languages[正则]

- **Regular expression**s are to describe all the languages that can be built from the operators applied to the symbols of some alphabet
- Regular Expression is a simple notation
  - Can express simple patterns (e.g., repeating sequences)
  - Not powerful enough to express English (or even C)
  - But powerful enough to express tokens (e.g., identifiers)
- Languages that can be expressed using regular expressions are called **Regular Languages**
- More complex languages need more complex notations
  - More complex languages and expressions[非正则] will be covered later

# Atomic REs[原子表达式]

- Atomic
  - Smallest RE that cannot be broken down further

- **Epsilon or ε** character denotes a zero length string
  - ε = {“”}
- **Single character** denotes a set of one string
  - ‘c’ = {“c”}
- Empty set is { } = ϕ, not the same as ε
  - Size(ϕ) = 0
  - Size(ε) = 1
  - Length(ε) = 0

# Compound REs[组合表达式]

- **Compound**
  - Large REs built from smaller ones
- Suppose *r* and *s* are REs denoting languages L(*r*) and L(*s*)
  - (*r*) is a RE denoting the language L(*r*)
    - We can add additional () around expressions without changing the language they denote
  - (*r*)|(*s*) is a RE denoting the language L(*r*) ∪ L(*s*)
  - (*r*)(*s*) is a RE denoting the language L(*r*)L(*s*)
  - $(r)^*$ is a RE denoting the language $(L(r))^*$
- REs often contain unnecessary (), which could be dropped
  - **(A) ≡ A**: A is a RE
  - (a)|((b)*(c)) ≡ a|b*c

中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle
# 编 译 原 理

## 第2讲：词法分析(2)

张献伟

xianweiz.github.io

DCS290, 3/5/2024

# Review Questions

- Q1: input and output of lexical analysis?

  Input: source code/char stream, output: tokens

- Q2: lexical analysis of "while i>=1)"?

  (keyword, 'while'), (id, 'i'), (sym, '>='), (num, '1'), (sym, ')')

- Q3: Σ = {a, b}, $L_1$ = {aa}, $L_2$={bbb}. What are $L_1 \mid L_2$ and $L_1L_2$?

  $L_3 = L_1 \mid L_2$ = {aa} | {bbb} = {aa, bbb}, $L_4 = L_1L_2$ = {aabbb}

- Q4: $L_3^2$?

  $L_3^2 = L_3L_3$ = {aa, bbb}{aa, bbb} = {aaaa, aabbb, bbbaa, bbbbbb}

- Q5: describe the meaning of $L_1^* \mid L_2^*$?

  A language composed of '*a*'s and 'b's of length 2N and 3N, respectively, including ε

- Q6: RE of identifiers in C language?

  (_letter)(_letter|digit)*

# Compound REs[组合表达式]

- **Compound**
  - Large REs built from smaller ones
- Suppose *r* and *s* are REs denoting languages L(*r*) and L(*s*)
  - (*r*) is a RE denoting the language L(*r*)
    - We can add additional () around expressions without changing the language they denote
  - (*r*)|(*s*) is a RE denoting the language L(*r*) ∪ L(*s*)
  - (*r*)(*s*) is a RE denoting the language L(*r*)L(*s*)
  - $(r)^*$ is a RE denoting the language $(L(r))^*$
- REs often contain unnecessary (), which could be dropped
  - **(A) ≡ A**: A is a RE
  - (a)|((b)*(c)) ≡ a|b*c

# Operator Precedence[优先级]

- RE operator precedence
  - $(A)$
  - $A^*$
  - $AB$
  - $A|B$

- Example: $ab^*c|d$
  - $a(b^*)c|d$
  - $(a(b^*))c|d$
  - $((a(b^*))c)|d$

# Common REs[常用表达]

- **At least one**: $A+ \equiv AA^*$
- **Option**: $A? \equiv A \mid \varepsilon$
- **Characters**: $[a_1a_2...a_n] \equiv a_1|a_2|...|a_n$
- **Range**: ‘a’ + ‘b’ + … + ‘z’ ≡ [a-z]
- **Excluded range**: complement of [a-z] ≡ [^a-z]

# RE Examples

| Regular Expression | Explanation |
|---|---|
| $a^*$ | 0 or more a's (ε, a, aa, aaa, aaaa, …) |
| a+ | 1 or more a's (a, aa, aaa, aaaa, …) |
| (a\|b)(a\|b) | (aa, ab, ba, bb) |
| $(a\|b)^*$ | all strings of a's and b's (including ε) |
| $(aa\|ab\|ba\|bb)^*$ | all strings of a's and b's of even length |
| [a-zA-Z] | shorthand for "a\|b\|…z\|A\|B\|…\|Z" |
| [0-9] | shorthand for "0\|1\|2\|…\|9" |
| $0([0-9])^*0$ | numbers that start and end with 0 |
| $1^*(0\|\varepsilon)1^*$ | binary strings that contain at most one zero |
| $(0\|1)^*00(0\|1)^*$ | all binary strings that contain '00' as substring |

- Q: are $(a|b)^*$ and $(a^*b^*)^*$ equivalent?

# Different REs of the Same Language

- $(a|b)^* = ?$
  - $(L(a|b))^* = (L(a) \cup L(b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$
  - $= \{a, b\}^0 + \{a, b\}^1 + \{a, b\}^2 + \ldots$
  - $= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$

- $(a^*b^*)^* = ?$
  - $(L(a^*b^*))^* = (L(a^*)L(b^*))^*$
  - $= L(\{\varepsilon, a, aa, \ldots\}\{\varepsilon, b, bb, \ldots\})^*$
  - $= L(\{\varepsilon, a, b, aa, ab, bb, \ldots\})^*$
  - $= \varepsilon + \{\varepsilon, a, b, aa, ab, bb, \ldots\} + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^2 + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^3 + \ldots$

# More Examples

- Keywords: ‘if’ or ‘else’ or ‘then’ or ‘for’ …
  - RE = ‘i’‘f’ + ‘e’‘l’‘s’‘e’ + … = ‘if’ + ‘else’ + ‘then’ + …

‘+’ == ‘|’

- Numbers: a non-empty string of digits
  - digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
  - integer = digit digit*
  - Q: is ‘000’ an integer?
- Identifier: strings of letters or digits, starting with a letter
  - letter = ‘a’ + ‘b’ + … ‘z’ + ‘A’ + ‘B’ + … + ‘Z’ = [a-zA-Z]
  - RE = letter(letter + digit)*
  - Q: is the RE valid for identifiers in C?
- Whitespace: a non-empty sequence of blanks, newline, tabs
  - (‘ ’ + ‘\n’ + ‘\t’)+

# REs in Programming Language

| Symbol | Meaning | | |
|---|---|---|---|
| **\d** | Any decimal digit, i.e. [0-9] | | |
| **\D** | Any non-digit char, i.e., [^0-9] | | |
| **\s** | Any whitespace char, i.e., [ \t\n\r\f\v] | | |
| **\S** | Any non-whitespace char, i.e., [^ \t\n\r\f\v] | | |
| **\w** | Any alphanumeric char, i.e., [a-zA-Z0-9_] | | |
| **\W** | Any non-alphanumeric char, i.e., [^a-zA-Z0-9_] | | |
| **.** | Any char | **\.** | Matching "." |
| **[a-f]** | Char range | **[^a-f]** | Exclude range |
| **^** | Matching string start | **$** | Matching string end |
| **(…)** | Capture matches | | |

https://docs.python.org/3/howto/regex.html

# Lexical Specification of a Language

- **S0**: write a regex for the lexemes of each token class
  - Numbers = digit+
  - Keywords = ‘if’ + ‘else’ + …
  - Identifiers = letter(letter + digit)*
- **S1**: construct R, matching all lexemes for all tokens
  - R = numbers + keywords + identifiers + … = $R_1 + R_2 + R_3 + \ldots$
- **S2**: let input be $x_q \ldots x_n$, for $1 \le i \le n$, check $x_1 \ldots x_i \in L(R)$
- **S3**: if successful, then we know $x_1 \ldots x_i \in L(R_j)$ for some j
  - E.g., an identifier or a number …
- **S4**: remove $x_1 \ldots x_i$ from input and go to step S2

# Lexical Spec. of a Language(cont.)

- How much input is used?
  - $x_1 \ldots x_i \in L(R)$, $x_1 \ldots x_j \in L(R)$, $i \neq j$
  - Which one do we want? (e.g., '==' or '=')
  - Maximal match: always choose the longer one[最长匹配]
- Which token is used if more than one matches?
  - $x_1 \ldots x_i \in L(R)$ where $R = R_1 + R_2 + \ldots + R_n$
  - $x_1 \ldots x_i \in L(R_m)$, $x_1 \ldots x_i \in L(R_n)$, $m \neq n$
    - E.g., keywords = 'if', identifier = letter(letter+digit)*
  - Keyword has higher priority
  - Rule of thumb: choose the one listed first[次序]
- What if no rule matches?
  - $x_1 \ldots x_i \notin L(R)$ → Error

# Summary: RE

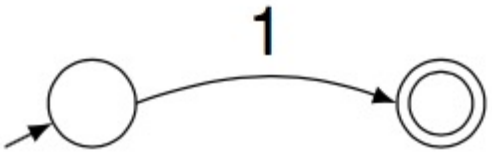
- We have learnt how to specify tokens for lexical analysis[定义]
  - Regular expressions
    - Concise notations for the string patterns
- Used in lexical analysis with some extensions[适度扩展]
  - To resolve ambiguities
  - To handle errors
- RE is only a language specification[只是定义了语言]
  - An implementation is still needed
  - Next: to construct a token recognizer for languages given by regular expressions – by using **finite automata**[有穷自动机]

# Impl. of Lexical Analyzer[实现]

- How do we go from specification to implementation?
  - RE → finite automata (FA)
- **Solution 1**: to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
  - Programmer specifies tokens using REs
  - The tool generates the source code from the given REs
    - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
- **Solution 2**: to write the code yourself
  - More freedom; even tokens not expressible through REs
  - But difficult to verify; not self-documenting; not portable; usually not efficient
  - ~~Generally not encouraged~~

# Transition Diagram[转换图]

- REs → transition diagrams
  - By hand
  - Automatic




- 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]: directed, labeled with 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

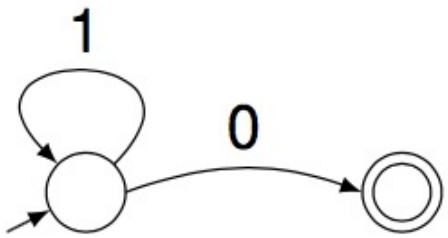
# FA: Language

- An FA is a program for classifying strings (accept, reject)
  - In other words, a program for recognizing a language
  - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
  - For a given string ‘x’, if there is transition sequence for ‘x’ to move from start state to 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)

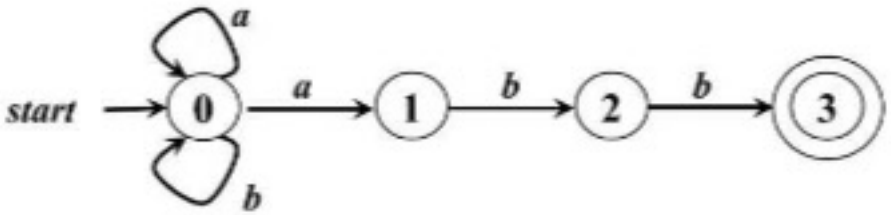
# Example

- 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




L(FA): all strings of ∑{a, b}, ending with 'abb'

L(RE) = (a|b)*abb

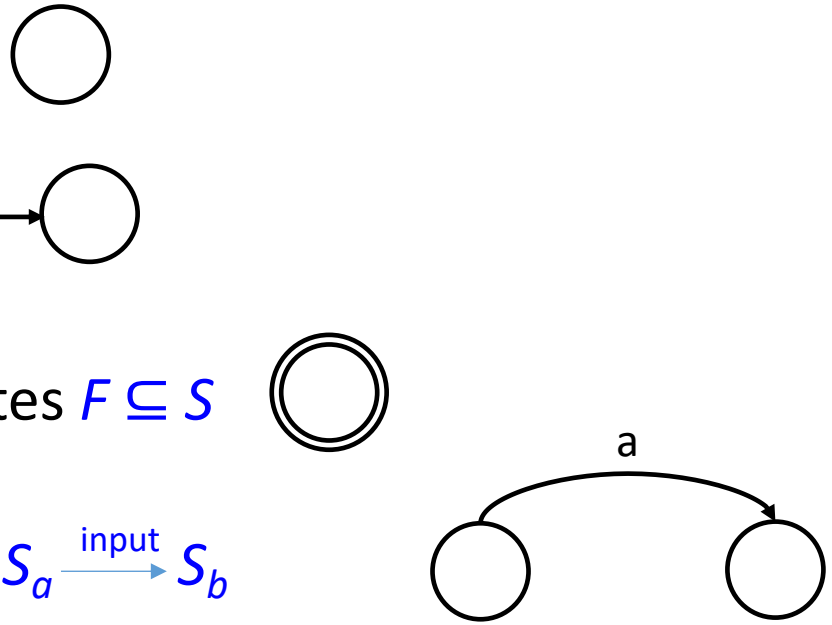
# DFA and NFA

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

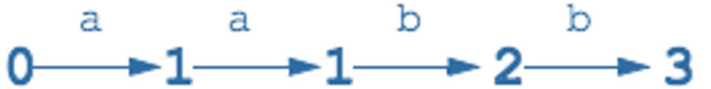
- Nondeterministic Finite Automata (**NFA**): the machine can exist in multiple states at the same time[非确定]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the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

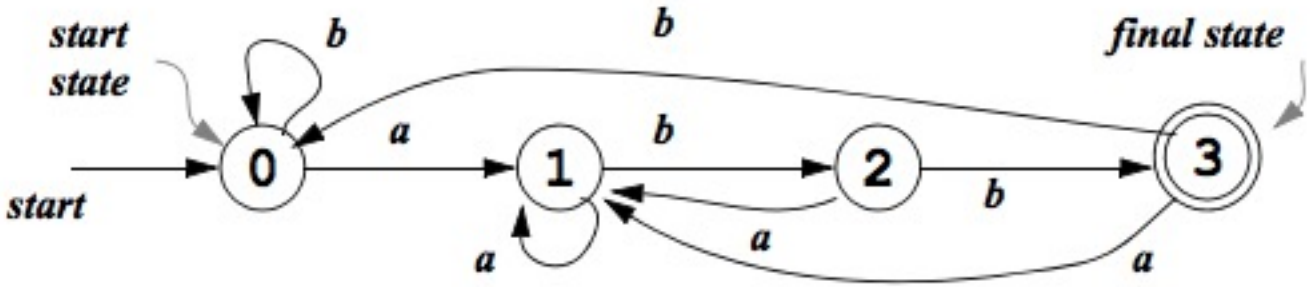
# State Graph

- 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: $S_a \xrightarrow{\text{input}} S_b$



# Example: DFA

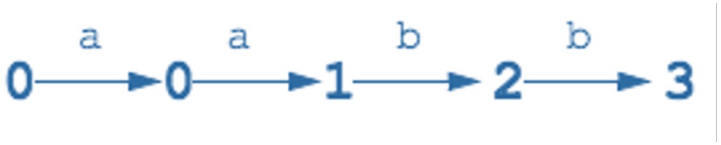
- There is **only one** possible sequence of moves --- either lead to a final state and accept or the input string is rejected
  - Input string: aabb
  - Successful sequence: 

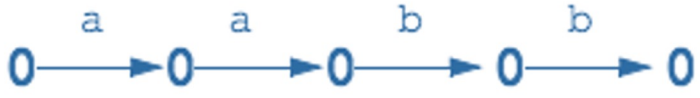

A DFA accepts $(a|b)^*abb$

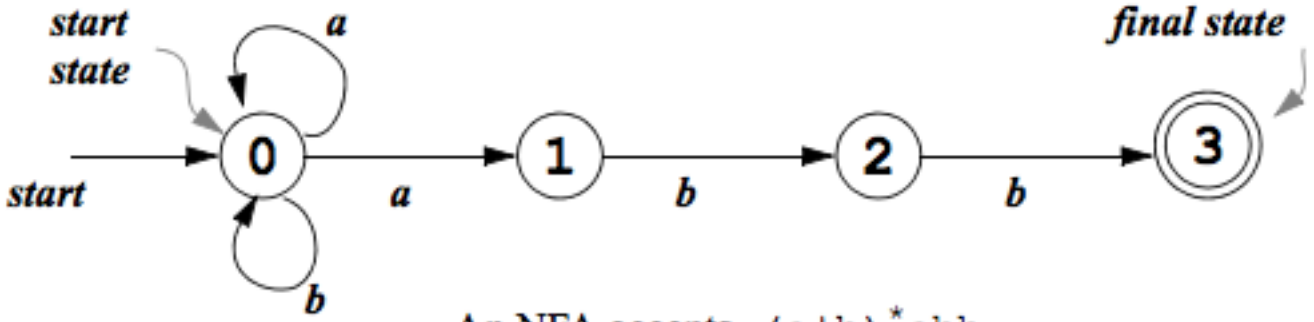
# Example: NFA

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




  - Unsuccessful sequence:

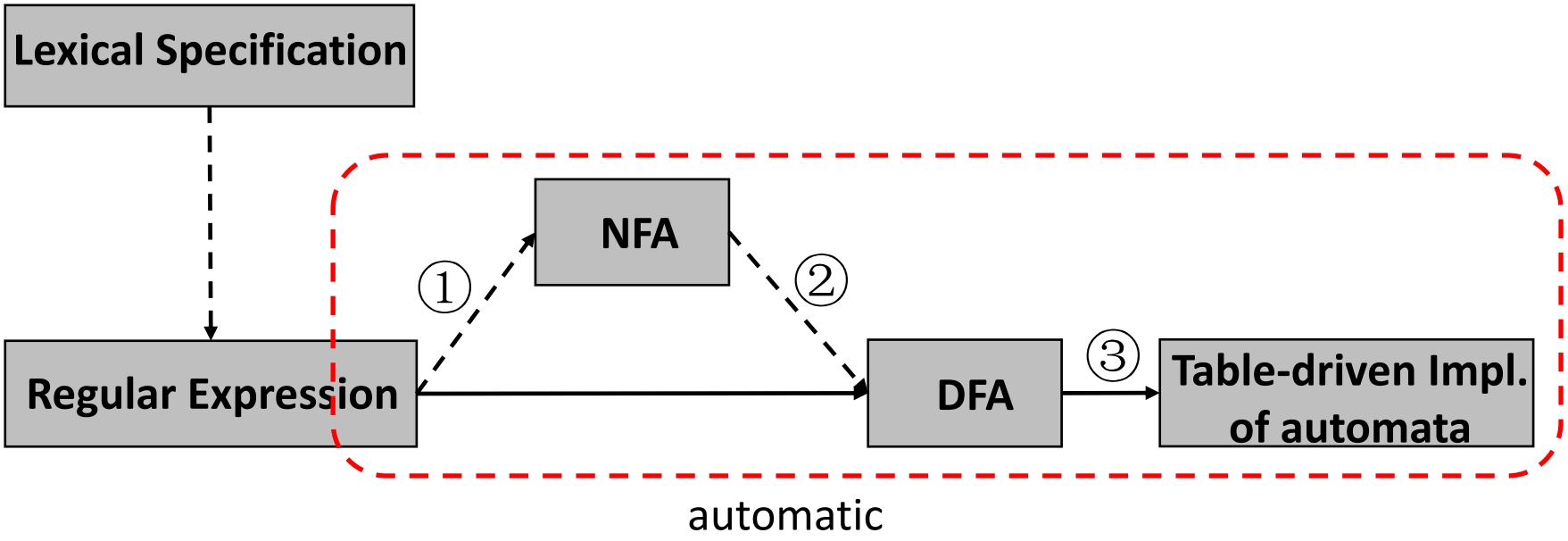



An NFA accepts $(a|b)^*abb$

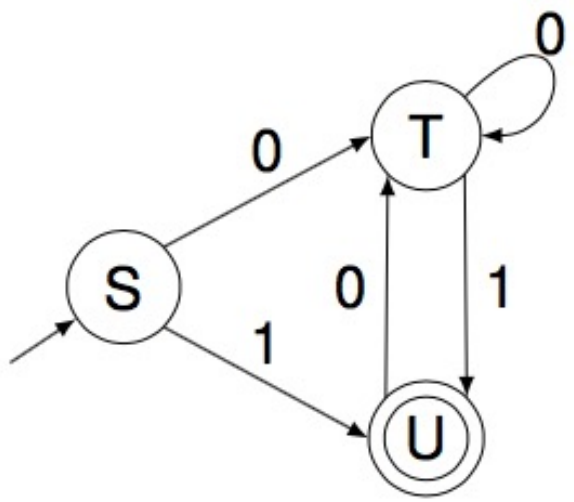
# Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



# DFA → Table

- FA can also be represented using transition table




alphabet →

state ↓

| | 0 | 1 |
|---|---|---|
| S | **T** | **U** |
| T | **T** | **U** |
| U | **T** | **x** |

**Table-driven Code:**

```
DFA() {
    state = "S";
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == "x")
            print("reject");
    }
    if (state ∈ F)
        printf("accept");
    else
        printf("reject");
}
```

Q: which is/are accepted?

111

000

001

# More on Table

- Implementation is efficient[表格是一种高效实现]
  - Table can be automatically generated
  - Need finite memory O(S x ∑)
    - Size of transition table
  - Need finite time O(input length)
    - Number of state transitions

- Pros and cons of table[表格实现的优劣]
  - Pro: can easily find the transitions on a given state and input
  - Con: 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

中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle
# 编 译 原 理

## 第3讲：词法分析(3)

张献伟

xianweiz.github.io

DCS290, 3/7/2024

# Review Questions

- Q1: RE of binary numbers that are multipliers of 2?

  (0|1)*0

- Q2: meaning of (a|b)*bb(a|b)*?

  Strings of a's and b's with consecutive b's

- Q3: usage of RE and FA in lexical analysis?

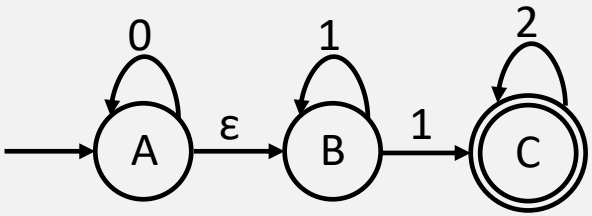
  RE: specify the token class; FA: implement the token recognizer

- Q4: general workflow from RE to implementation?

  RE → NFA → DFA → Table

- Q5: the graph describes NFA or DFA? Why?

  NFA. A: ε-transition, B: 1-transition



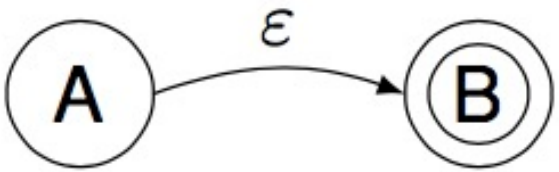
# More on Table

- Implementation is efficient[表格是一种高效实现]
  - Table can be automatically generated
  - Need finite memory O(S x ∑)
    - Size of transition table
  - Need finite time O(input length)
    - Number of state transitions

- Pros and cons of table[表格实现的优劣]
  - Pro: can easily find the transitions on a given state and input
  - Con: 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

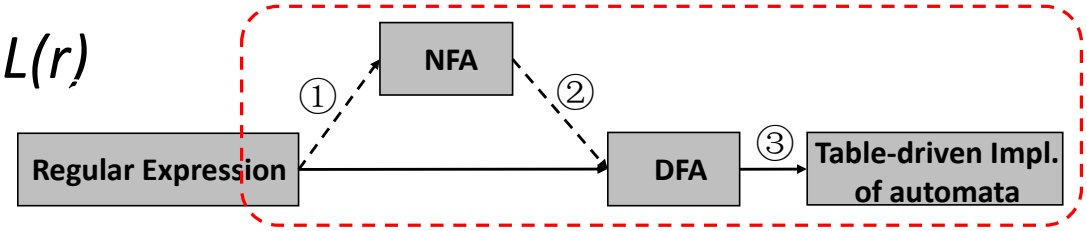
# RE → NFA

- NFA can have ε-moves
  - Edges labelled with ε
  - Move from state *A* to state *B* without reading any input



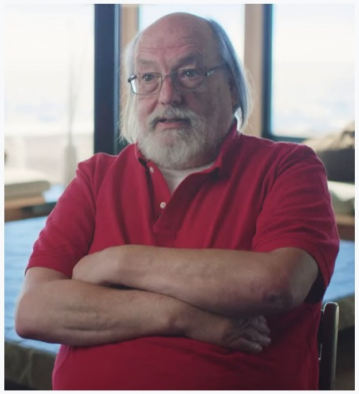

- **M-Y-T algorithm** (Thompson's construction) to convert any RE to an NFA that defines the same language[正则表达式转换到自动机]
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

**McNaughton-Yamada-Thompson**



# Thompson

Ken Thompson



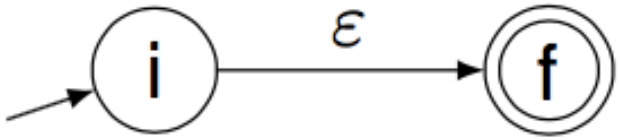
**Kenneth Lane Thompson** (born February 4, 1943) is an American pioneer of computer science & Computer Chess Development. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating system. Since 2006, Thompson has worked at Google, where he co-developed the Go programming language.

Other notable contributions included his work on regular expressions and early computer text editors QED and ed, the definition of the UTF-8 encoding, and his work on computer chess that included the creation of endgame tablebases and the chess machine Belle. He won the Turing Award in 1983 with his long-term colleague Dennis Ritchie.

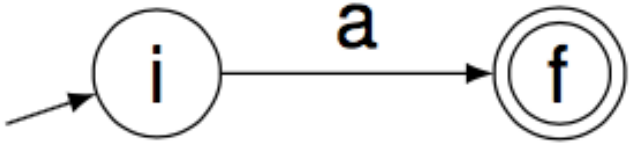
In the 1960s, Thompson also began work on regular expressions. Thompson had developed the CTSS version of the editor QED, which included regular expressions for searching text. QED and Thompson's later editor ed (the standard text editor on Unix) contributed greatly to the eventual popularity of regular expressions, and regular expressions became pervasive in Unix text processing programs. Almost all programs that work with regular expressions today use some variant of Thompson's notation. He also invented Thompson's construction algorithm used for converting regular expressions into nondeterministic finite automata in order to make expression matching faster.[12]

# RE → NFA (cont.)

- Step 1: processing atomic REs
  - ε expression[空]
    - *i* is a new state, the start state of NFA
    - *f* is another new state, the accepting state of NFA

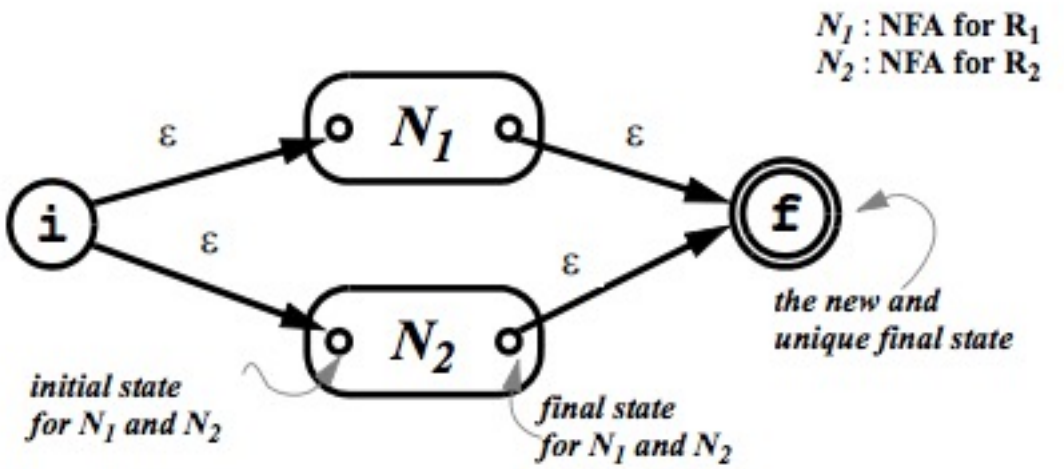



  - Single character RE *a*[单字符]

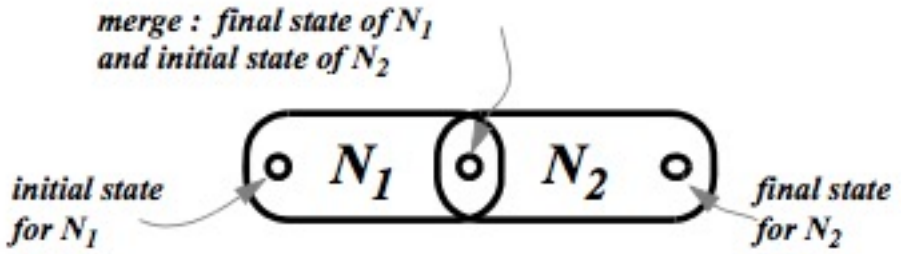


# RE → NFA (cont.)

- Step 2: processing compound REs[组合]
  - $R = R_1 \mid R_2$

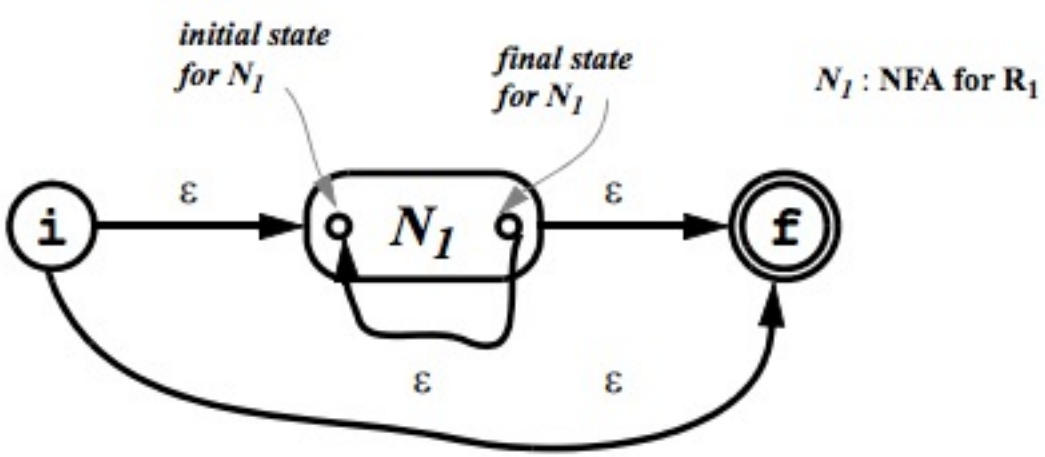



  - $R = R_1R_2$



# RE → NFA (cont.)

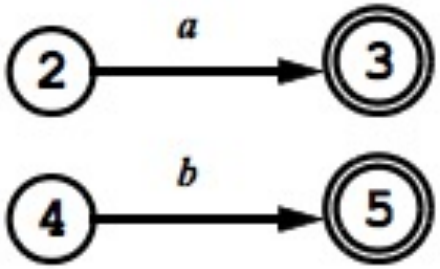
- Step 2: processing compound REs
  - $R = R_1^*$



# Example

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

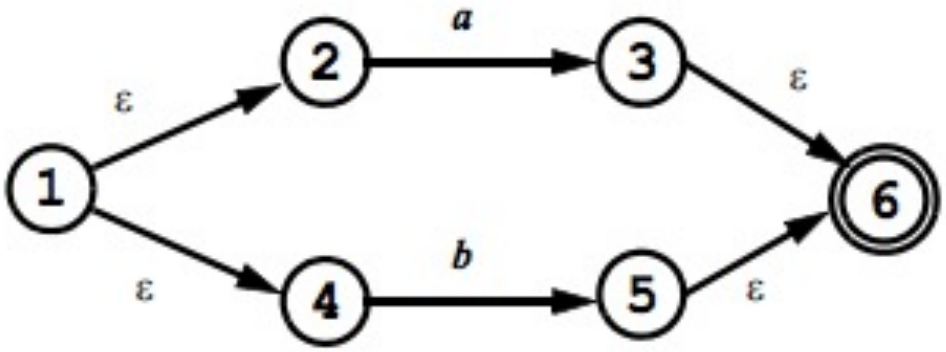
a (in a|b) ===>




b (in a|b) ===>

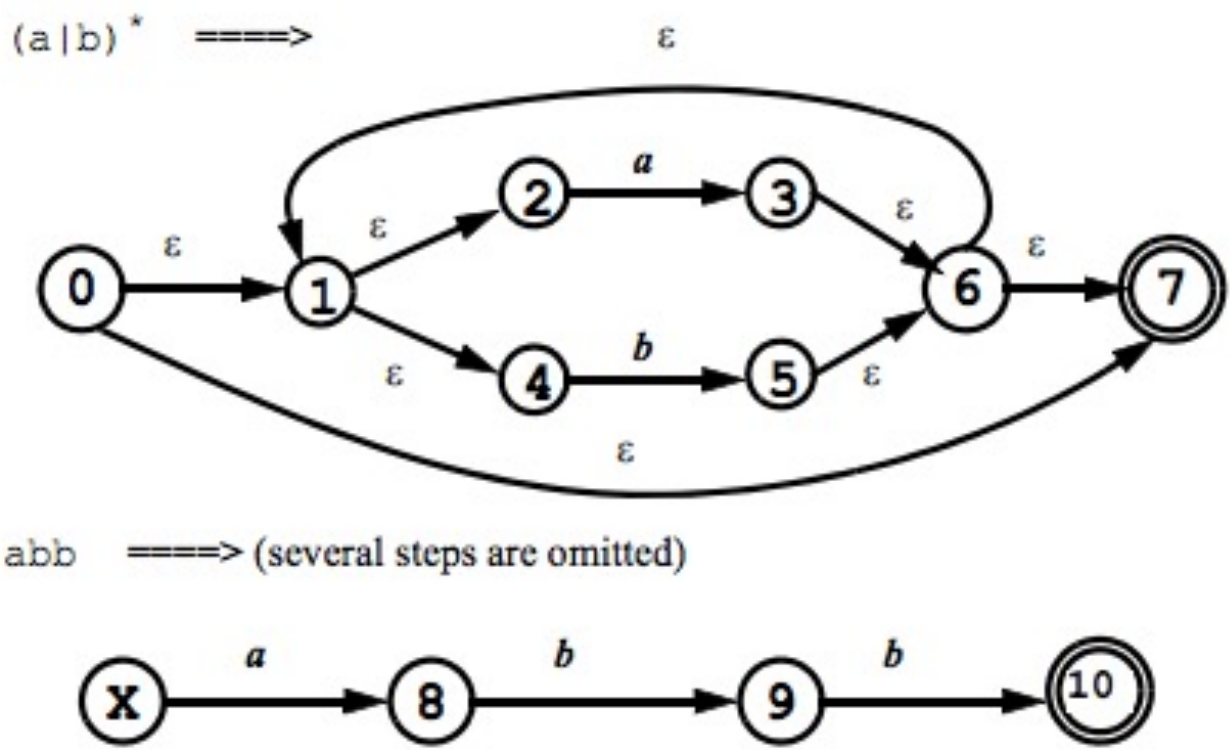



a|b ====>



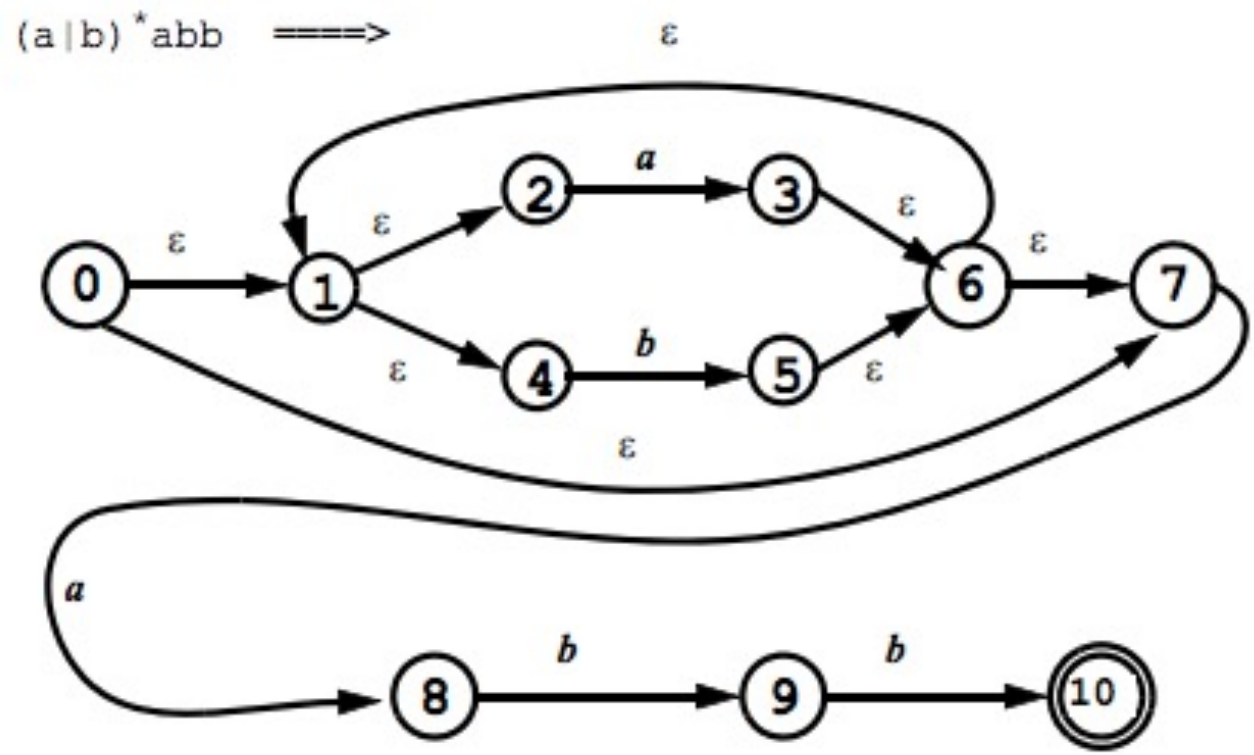
# Example (cont.)

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



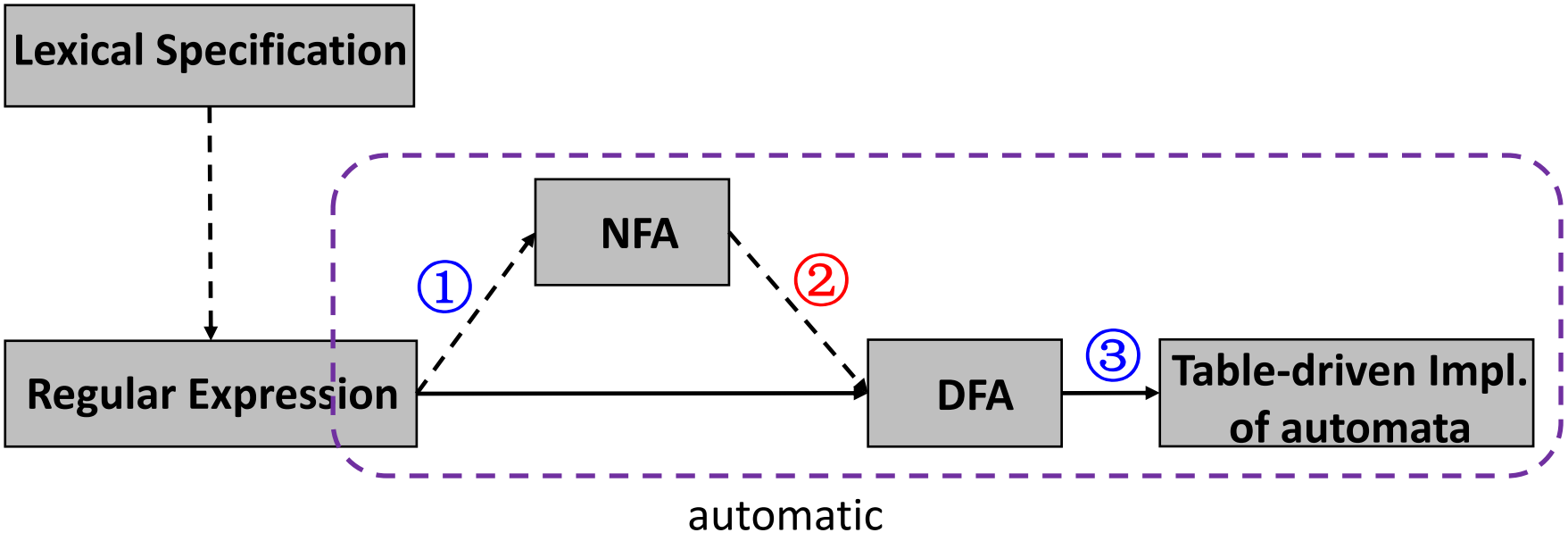
# Example (cont.)

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



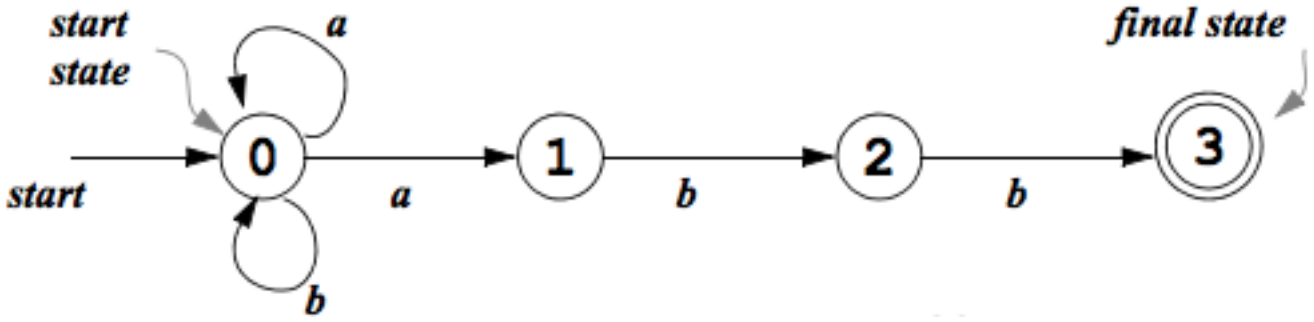
# The Conversion Flow[转换流程]

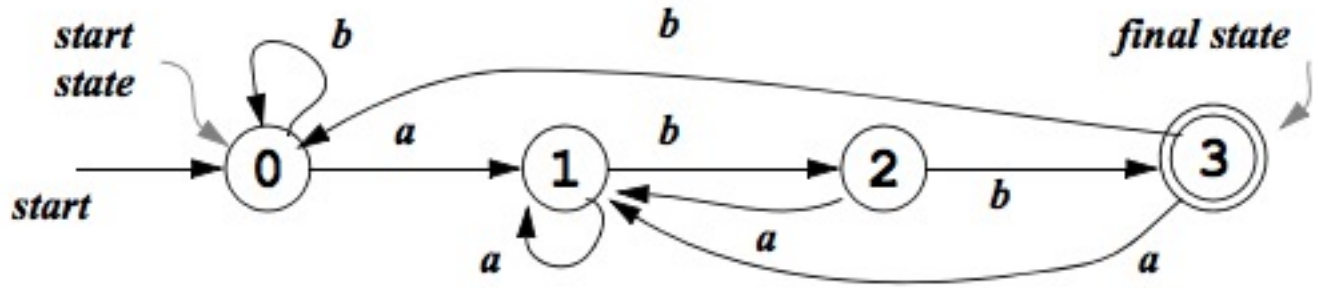
- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



# NFA → DFA: Same[等价]

- 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

# NFA → DFA: Theory[相关理论]

- Question: is L(NFA) ⊆ L(DFA)?
  - Otherwise, conversion would be futile
- Theorem: L(NFA) ≡ L(DFA)
  - Both recognize regular languages L(RE)
  - Will show L(NFA) ⊆ L(DFA) by construction (NFA → DFA)
  - Since L(DFA) ⊆ L(NFA), L(NFA) ≡ L(DFA)

Any DFA can be easily changed into NFA (e.g., add ε moves)

- 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 done because the speed of DFA far outweighs its extra memory consumption

# NFA → DFA: Idea

- Algorithm to convert[转换算法]
  - Input: an NFA *N*
  - Output: a DFA *D* accepting the same language as *N*

- **Subset construction**[子集构建]
  - Each state of the constructed DFA corresponds to a set of NFA states[一个DFA状态对应多个NFA状态]
    - Hence, the name 'subset construction'
  - After reading input $a_1a_2…a_n$, 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 $a_1a_2…a_n$

# NFA → DFA: Steps

- The **initial state** of the DFA is the set of all states the NFA can be in without reading any input[初始状态]
- For any state $\{q_i , q_j , \dots , q_k\}$ of the DFA and any input *a*, the **next state** of the DFA is the set of all states of the NFA that can result as next states if the NFA is in any of the states $q_i , q_j , \dots , q_k$ when it reads *a*[下一状态]
  - This includes states that can be reached by reading *a* followed by any number of ε-transitions
  - Use this rule to keep adding new states and transitions until it is no longer possible to do so
- The **accepting states** of the DFA are those states that contain an accepting state of the NFA[接收状态]

# NFA → DFA: Algorithm

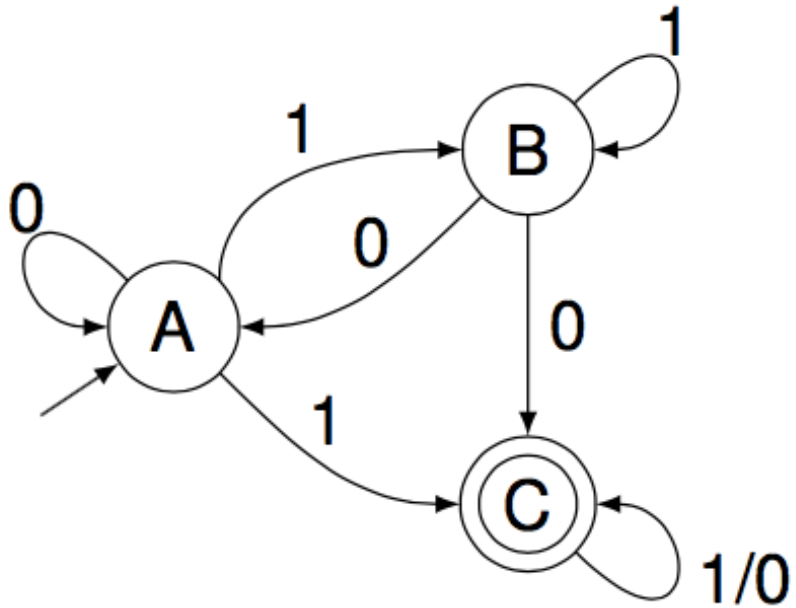
Initially, ε-*closure*($s_0$) is the only state in *Dstates* and it is unmarked
**while** there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    **for** each input symbol $a \in \Sigma$ **do**
        $U$ := ε-*closure*(*move*(*T*,*a*))
        **if** *U* is not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        **end if**
        *Dtran*[*T*,*a*] := *U*
    **end do**
**end do**

- Operations on NFA states:
  - 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**; = $\cup_{s \text{ in } T}$ε-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 → DFA: Example

- Start by constructing ε-closure of the start state[初始状态]
  - ε-closure(A) = A
- Keep getting ε-closure(*move(T, a)*)[更多状态]
- Stop, when there are no more new states

***T***: A, ***a***: 0/1



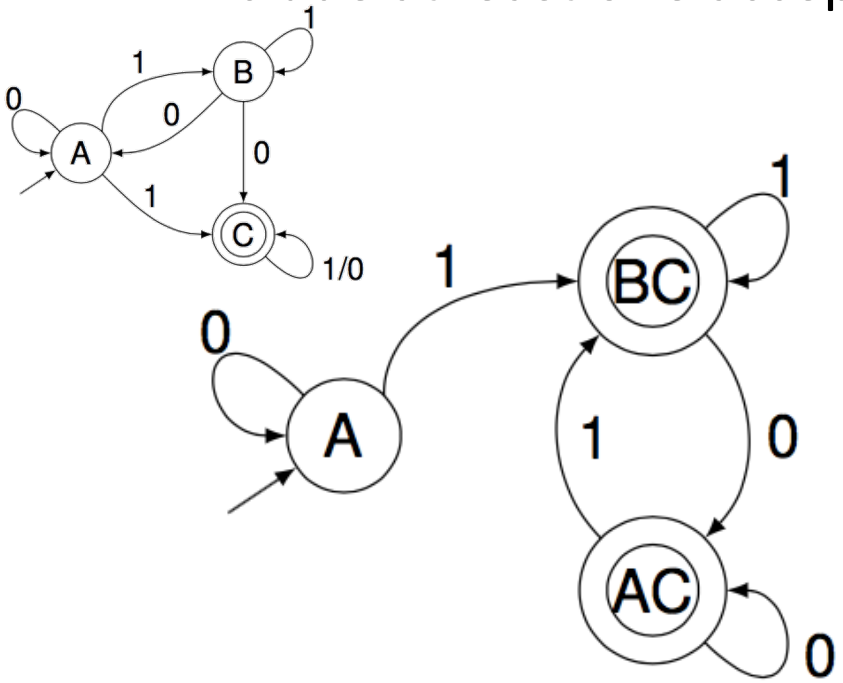

alphabet

state

| | 0 | 1 |
|---|---|---|
| A | A | BC |
| BC | AC | BC |
| AC | AC | BC |

# NFA → DFA: Example (cont.)

- Mark the final states of the DFA[终止状态]
  - The accepting states of *D* are all those sets of *N*'s states that include at least one accepting state of *N*




alphabet

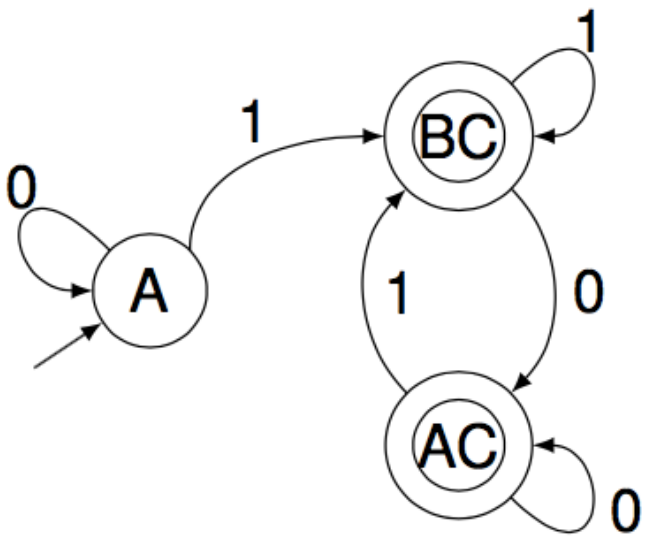
state

| | 0 | 1 |
|---|---|---|
| A | **A** | **BC** |
| BC | **AC** | **BC** |
| AC | **AC** | **BC** |

- Is the DFA minimal?
  - As few states as possible

# NFA → DFA: Minimization[最小化]

- Any DFA can be converted to its minimum-state equivalent DFA
  - Discover sets of equivalent states[存在等价/重复状态]
  - Represent each such set with just one state
- Two states are equivalent if and only if:
  - $\forall \alpha \in \Sigma$, transitions on $\alpha$ lead to equivalent states
  - $\alpha$-transitions to distinct sets ⇒ states must be in distinct sets




Initial: {A}, {BC, AC}

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

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'

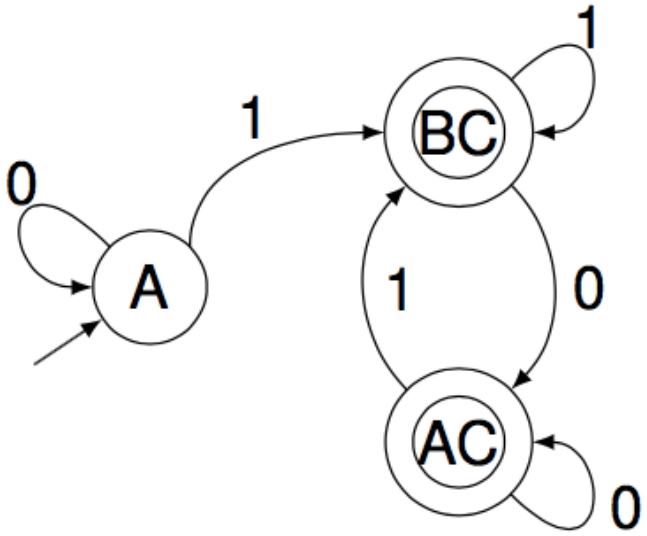
Final: {A}, {BCAC}



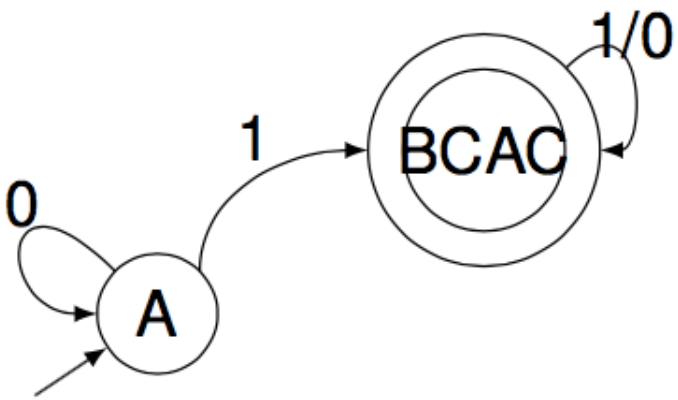
https://people.cs.umass.edu/~moss/610-slides/06.pdf

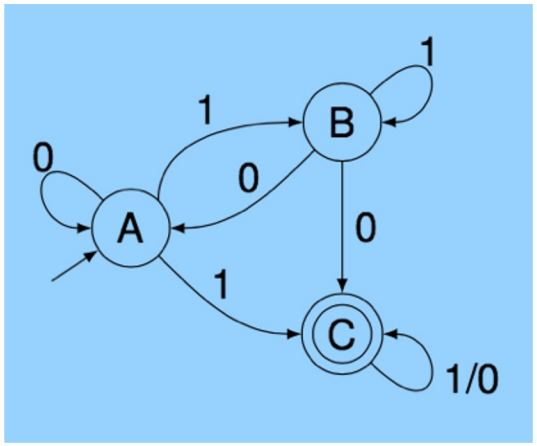
# NFA → DFA: Minimization (cont.)

- States *BC* and *AC* do not need differentiation
  - Should be merged into one




Minimized




| | 0 | 1 |
|---|---|---|
| A | A | BC |
| BC | AC | BC |
| AC | AC | BC |

# Minimization Algorithm

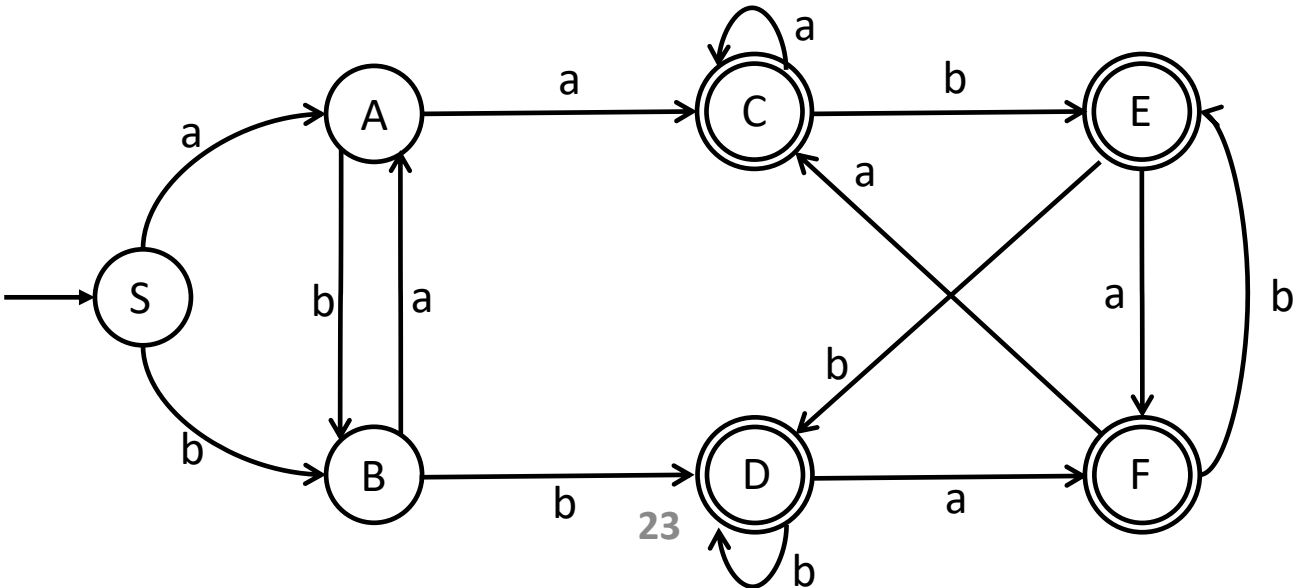
- The algorithm
  - Partitioning the states of a DFA into groups of states that cannot be distinguished (i.e., equivalent)
  - Each groups of states is then merged into a single state of the min-state DFA

```
P <- {F}, {S - F}
while (P is still changing)
  T <- { }
  for each state s ∈ P
    for each α ∈ Σ
      partition s by α into s1 & s2
      T <- T ∪ s1 ∪ s2
  if T ≠ P then
    P <- T
```

- For a DFA（∑, S, n, F, $\delta$）
  - The initial partition $P_0$, has two sets and {*S* - *F*}
  - Splitting a set (i.e., partitioning a set by input symbol $\alpha$)
    - Assume $q_a$ and $q_b \in$ ***S***, and $\delta(q_a, \alpha) = q_x$ and $\delta(q_b, \alpha) = q_y$
    - If $q_x$ and $q_y$ are not in the same set, then ***S*** must be split (i.e., $\alpha$ splits ***S***)
    - One state in the final DFA cannot have two transitions on $\alpha$

# Example

- P0: $s_1$ = {S, A, B}, $s_2$ = {C, D, E, F}
- For $s_1$, further splits into {S}, {A}, {B}
  - a: S --> A ∈ $s_1$, A --> C ∈ $s_2$, B --> A ∈ $s_1$ ⟹ a distincts $s_1$ => {S, B}, {A}
  - b: S --> B ∈ $s_1$, A --> B ∈ $s_1$, B --> D ∈ $s_2$ ⟹ b distincts $s_1$ => {S}, {B}, {A}
- For $s_2$, all states are equivalent
  - a: C --> C ∈ $s_2$, D --> F ∈ $s_2$, E --> F ∈ $s_2$ , F --> C ∈ $s_2$ ⟹ a doesn't
  - b: C --> E ∈ $s_2$, D --> D ∈ $s_2$, E --> D ∈ $s_2$ , F --> E ∈ $s_2$ ⟹ b doesn't

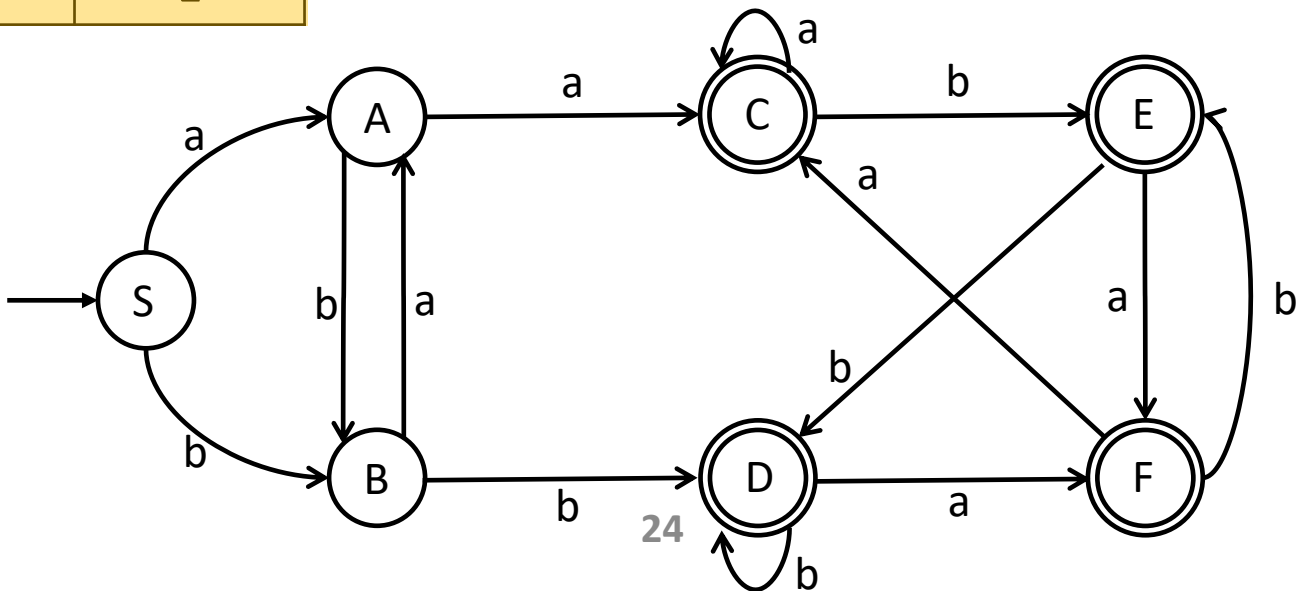


# Example (cont.)

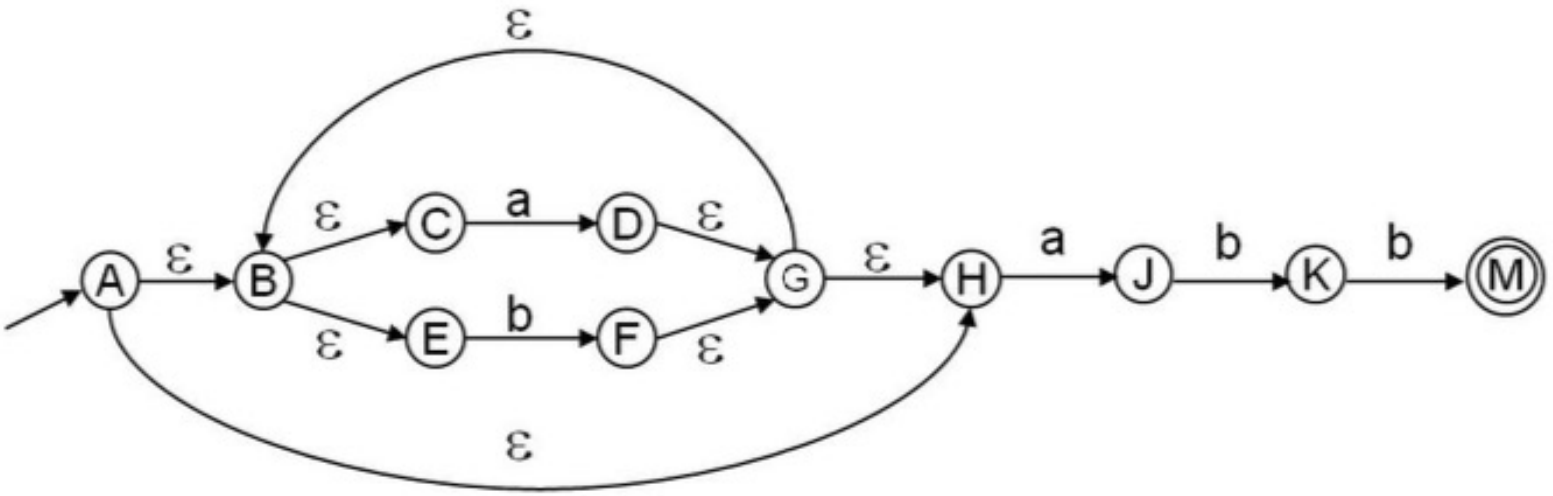
| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **C** | C | E |
| **D** | F | D |
| **E** | F | D |
| **F** | C | E |

| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **CF** | C | E |
| **DE** | F | D |

| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **CFDE** | CF | DE |



# NFA → DFA: More Example

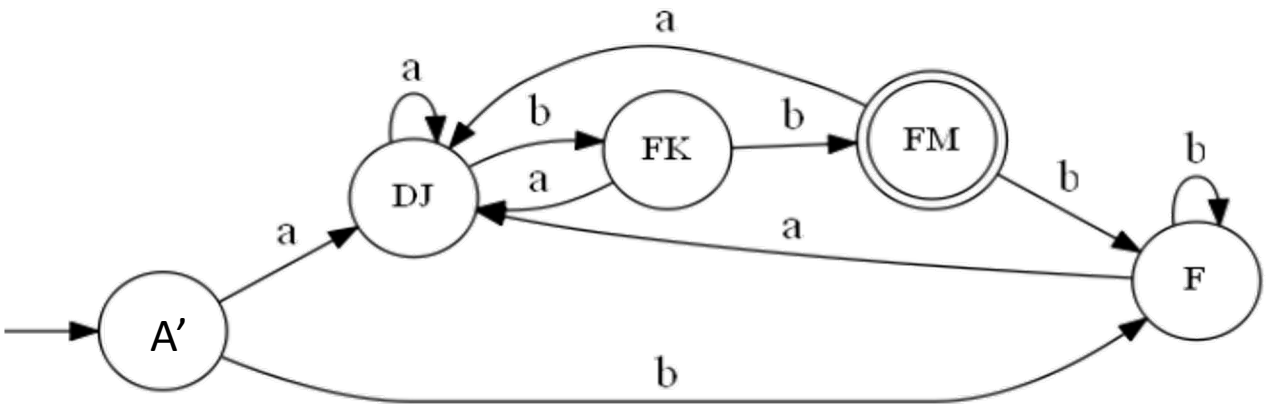



- Start state of the equivalent DFA
  - ε-closure(A) = {A, B, C, E, H} = A'
- ε-closure(move(A', a)) = ε-closure({D, J}) = {B, C, D, E, H, G, J} = B'
- ε-closure(move(A', b)) = ε-closure({F}) = {B, C, E , F, G, H} = C'
- … …

# NFA → DFA: More Example (cont.)

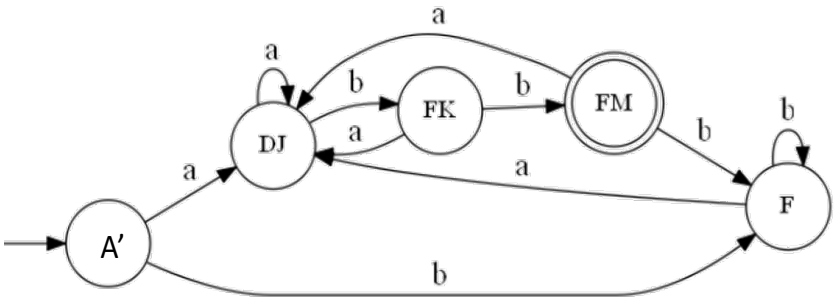
| | a | b |
|---|---|---|
| A’ | DJ | F |
| DJ | DJ | FK |
| F | DJ | F |
| FK | DJ | FM |
| FM | DJ | F |

- Is the DFA minimal?
  - States A’ and F should be merged
- Should we merge states A’ and FM?
  - NO. A’ and FM are in different sets from the very beginning (FM is accepting, A’ is not).



# NFA → DFA: More Example (cont.)

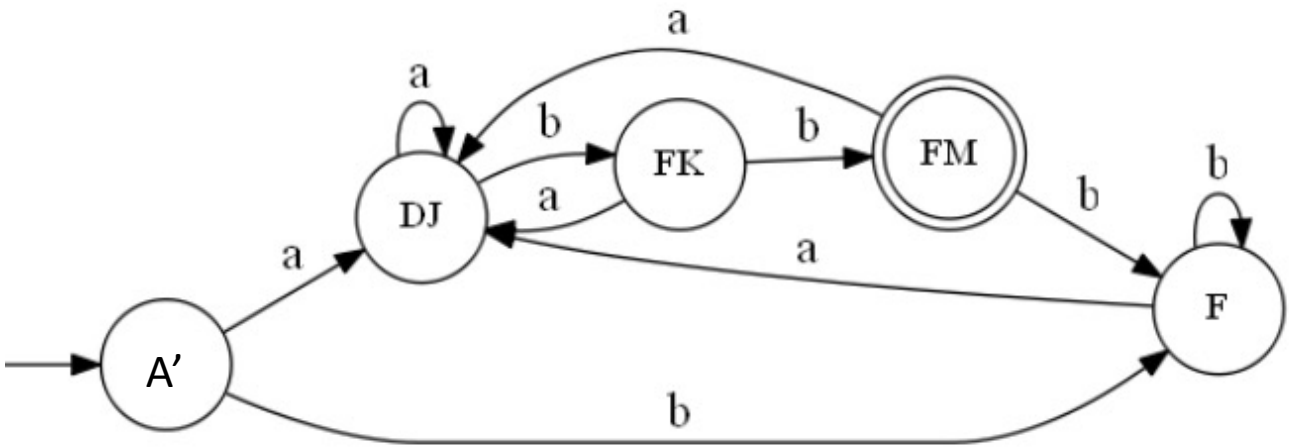
- P0: $s_1$ = {A', DJ, FK, F}, $s_2$ = {FM}
- For $s_1$, further splits into {A', DJ, F}, {FK}
  - a: A'--> DJ ∈ $s_1$, DJ --> DJ ∈ $s_1$, FK --> DJ ∈ $s_1$, F --> DJ ∈ $s_1$ ⟹ a doesn't distinct
  - b: A' --> F ∈ $s_1$, DJ --> FK ∈ $s_1$, FK --> FM ∈ $s_2$, F --> F ∈ $s_1$ ⟹ b distincts $s_1$ => $s_{11}$={A', DJ, F}, $s_{12}$={FK}
- For $s_{11}$, further splits into {A', DJ, F}, {FK}
  - a: A'--> DJ ∈ $s_{11}$, DJ --> DJ ∈ $s_{11}$, F --> DJ ∈ $s_{11}$ ⟹ a doesn't distinct
  - b: A'--> F ∈ $s_{11}$, DJ --> FK ∈ $s_{12}$, F --> DJ ∈ $s_{11}$ ⟹ b distincts $s_{11}$ => $s_{111}$={A', F}, $s_{112}$={DJ}
- For $s_{111}$, impossible to further split
- Final states: $S_{111}$ = {A', F}, $S_{112}$ = {DJ}, $S_{12}$ = {FK}, $S_2$ = {FM}



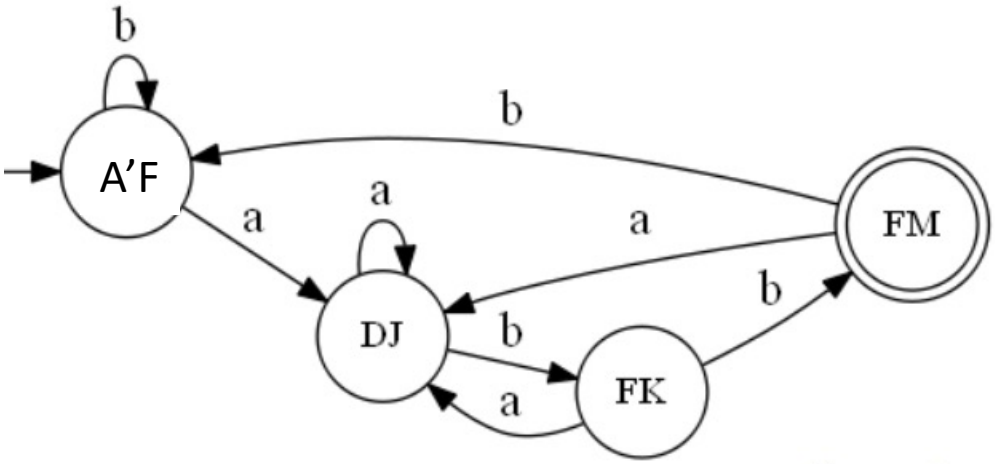

| | a | b |
|---|---|---|
| A' | DJ | F |
| DJ | DJ | FK |
| F | DJ | F |
| FK | DJ | FM |
| FM | DJ | F |

# NFA → DFA: More Example (cont.)

- Original DFA: before merging A’ and F




- Minimized DFA: Do you see the original RE (a|b)*abb



# NFA → DFA: Space Complexity[空间复杂度]

- 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

中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Compilation Principle
# 编 译 原 理

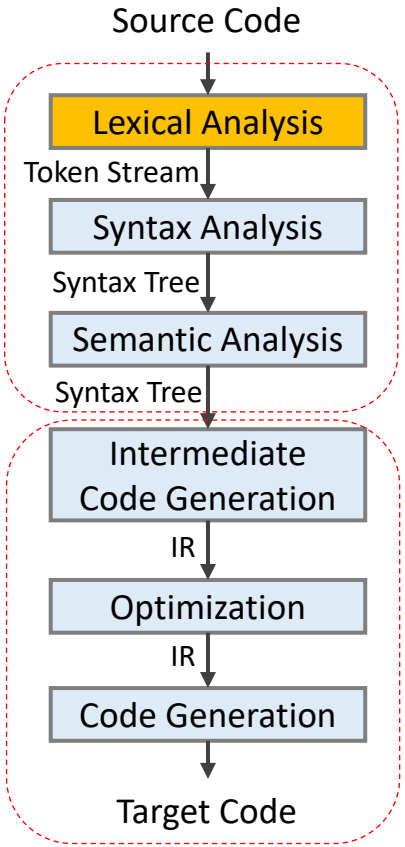
## 第4讲：词法分析(4)

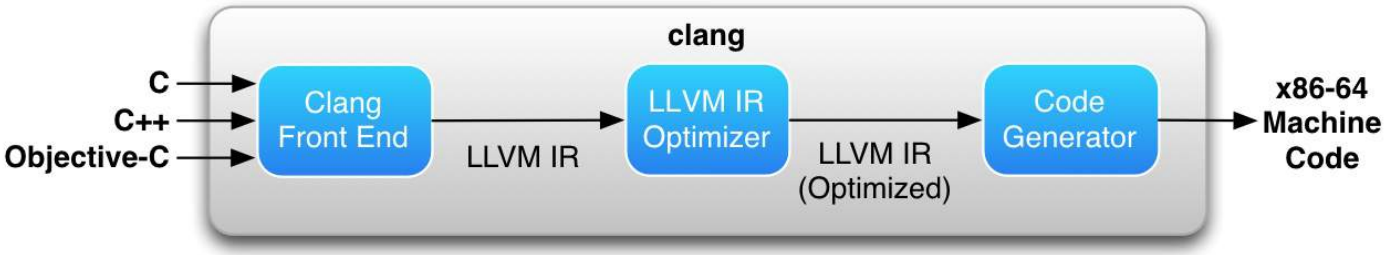
张献伟

xianweiz.github.io

DCS290, 3/12/2024

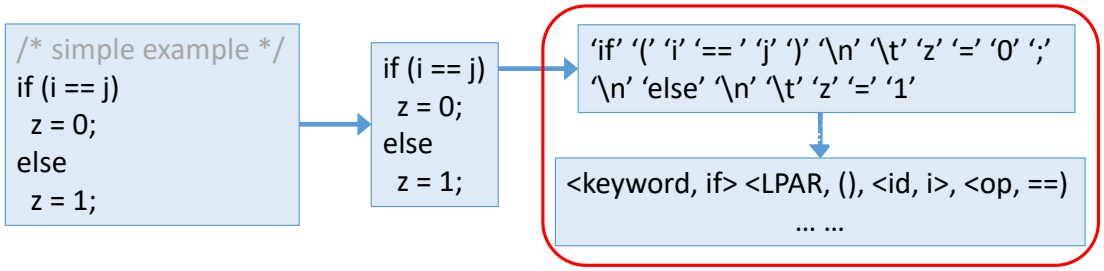
# Structure of a Typical Compiler[结构]

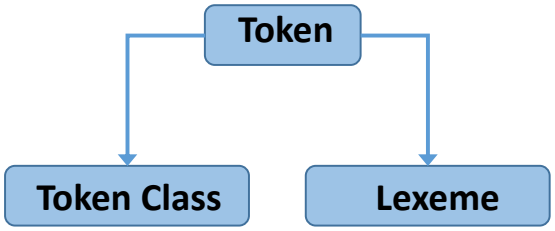


# Lexical Analysis

- Workflow
  - Partition the input character stream to lexemes
  - Identify the token class of each lexeme
- **Regular Expression** is a good way to specify tokens
  - Simple yet powerful (able to express patterns)
- **Finite Automata** is to construct a token recognizer for languages given by regular expressions
  - A program for classifying tokens (accept, reject)

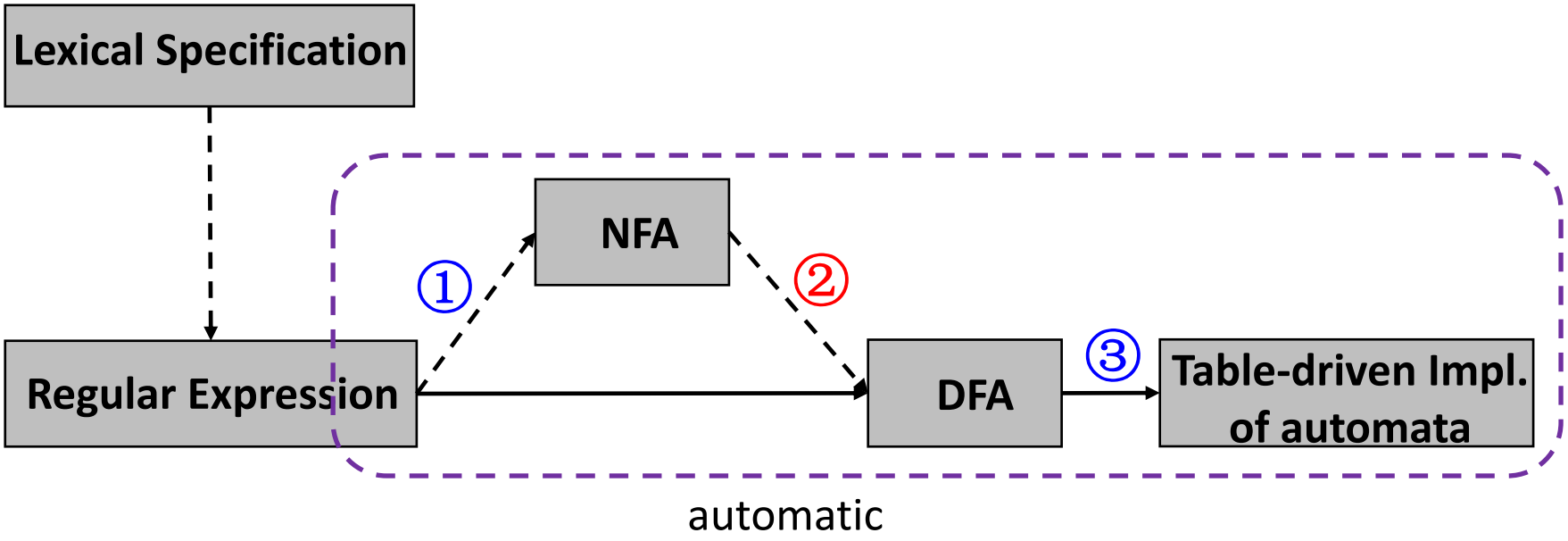



String patterns describing the class

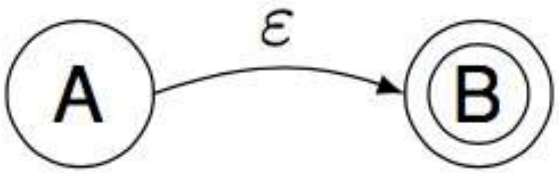
# The Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



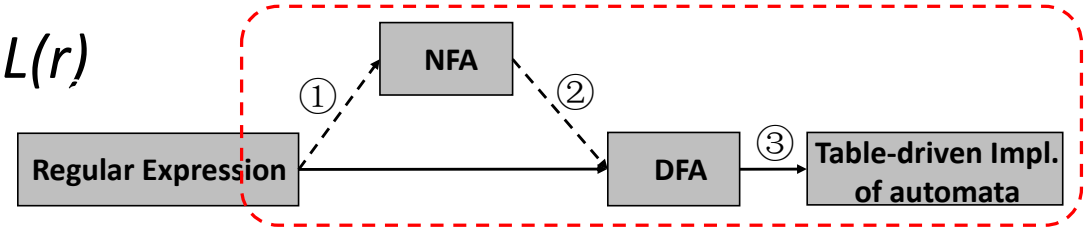
# RE → NFA

- NFA can have ε-moves
  - Edges labelled with ε
  - Move from state *A* to state *B* without reading any input



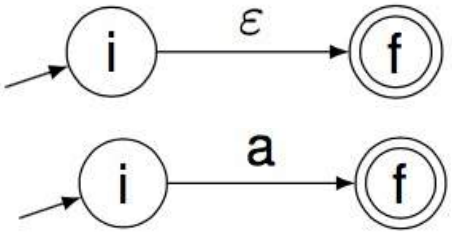

- **M-Y-T algorithm** (Thompson's construction) to convert any RE to an NFA that defines the same language[正则表达式转换到自动机]
  - Input: RE *r* over alphabet ∑
  - Output: NFA accepting *L(r)*

**McNaughton-Yamada-Thompson**

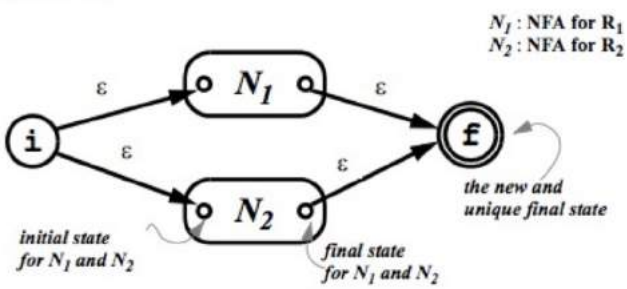


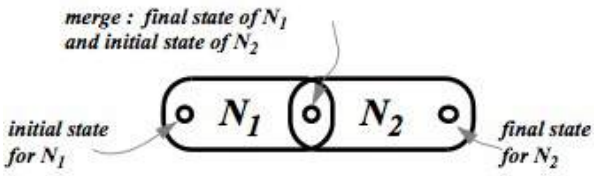
# RE → NFA (cont.)

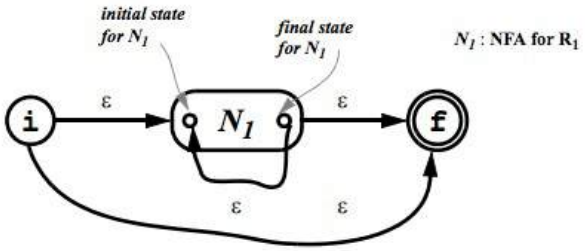
- Step 1: processing atomic REs
  - ε expression[空]
    - *i* is a new state, the start state of NFA
    - *f* is another new state, the accepting state of NFA
  - Single character RE *a*[单字符]




- Step 2: processing compound REs[组合]
  - $R = R_1 \mid R_2$
  - $R = R_1 R_2$
  - $R = R_1^*$



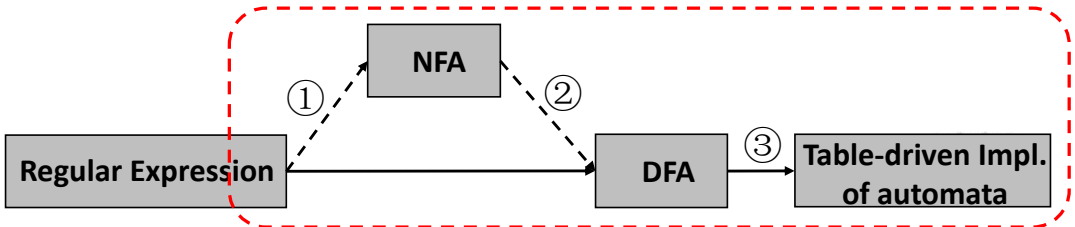



# NFA → DFA: Idea

- Algorithm to convert[转换算法]
  - Input: an NFA *N*
  - Output: a DFA *D* accepting the same language as *N*

- **Subset construction**[子集构建]
  - Each state of the constructed DFA corresponds to a set of NFA states[一个DFA状态对应多个NFA状态]
    - Hence, the name 'subset construction'
  - After reading input $a_1a_2...a_n$, 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 $a_1a_2...a_n$



# NFA → DFA: Steps

- The **initial state** of the DFA is the set of all states the NFA can be in without reading any input[初始状态]
- For any state $\{q_i , q_j , \dots , q_k\}$ of the DFA and any input *a*, the **next state** of the DFA is the set of all states of the NFA that can result as next states if the NFA is in any of the states $q_i , q_j , \dots , q_k$ when it reads *a*[下一状态]
  - This includes states that can be reached by reading *a* followed by any number of ε-transitions
  - Use this rule to keep adding new states and transitions until it is no longer possible to do so
- The **accepting states** of the DFA are those states that contain an accepting state of the NFA[接收状态]

# NFA → DFA: Algorithm

Initially, ε-*closure*($s_0$) is the only state in *Dstates* and it is unmarked
**while** there is an unmarked state *T* in *Dstates* **do**
    mark *T*
    **for** each input symbol $a \in \Sigma$ **do**
        $U$ := ε-*closure*(*move*(*T*,*a*))
        **if** *U* is not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*
        **end if**
        *Dtran*[*T*,*a*] := *U*
    **end do**
**end do**

- Operations on NFA states:
  - 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**; = $\cup_{s \text{ in } T}$ε-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***

# Minimization Algorithm

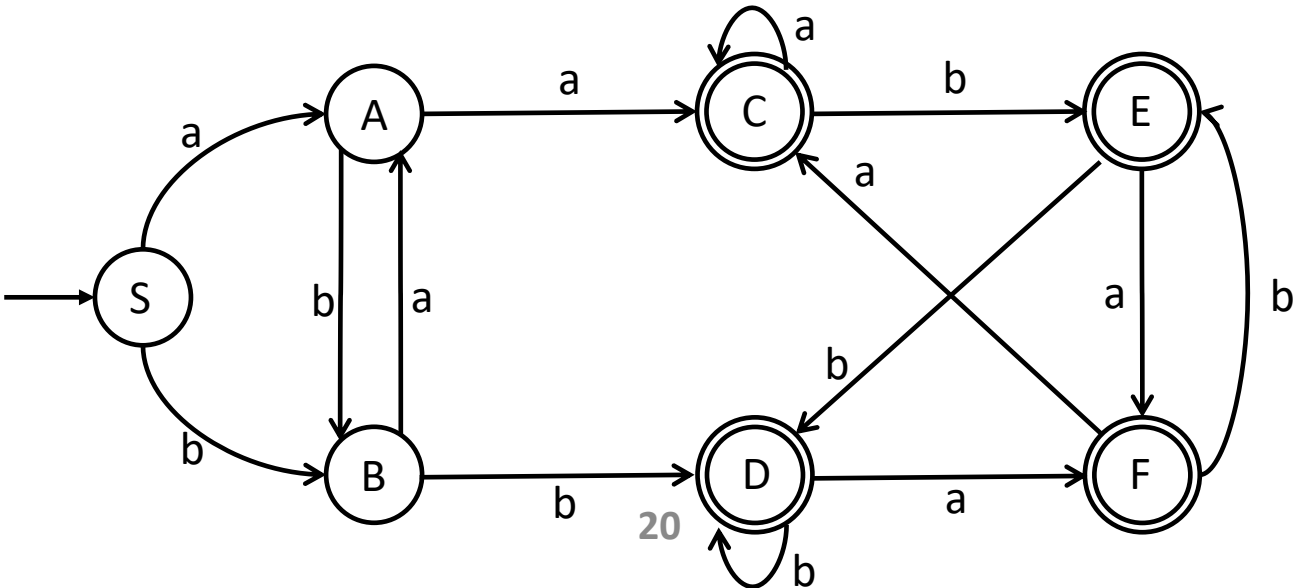
- The algorithm
  - Partitioning the states of a DFA into groups of states that cannot be distinguished (i.e., equivalent)
  - Each groups of states is then merged into a single state of the min-state DFA

```
P <- {F}, {S - F}
while (P is still changing)
  T <- { }
  for each state s ∈ P
    for each α ∈ Σ
      partition s by α into s1 & s2
      T <- T ∪ s1 ∪ s2
  if T ≠ P then
    P <- T
```

- For a DFA（∑, S, n, F, $\delta$）
  - The initial partition $P_0$, has two sets and {*S* - *F*}
  - Splitting a set (i.e., partitioning a set by input symbol $\alpha$)
    - Assume $q_a$ and $q_b \in$ ***S***, and $\delta(q_a, \alpha) = q_x$ and $\delta(q_b, \alpha) = q_y$
    - If $q_x$ and $q_y$ are not in the same set, then ***S*** must be split (i.e., $\alpha$ splits ***S***)
    - One state in the final DFA cannot have two transitions on $\alpha$

# Example

- P0: $s_1$ = {S, A, B}, $s_2$ = {C, D, E, F}
- For $s_1$, further splits into {S}, {A}, {B}
  - a: S --> A ∈ $s_1$, A --> C ∈ $s_2$, B --> A ∈ $s_1$ ⟹ a distincts $s_1$ => {S, B}, {A}
  - b: S --> B ∈ $s_1$, A --> B ∈ $s_1$, B --> D ∈ $s_2$ ⟹ b distincts $s_1$ => {S}, {B}, {A}
- For $s_2$, all states are equivalent
  - a: C --> C ∈ $s_2$, D --> F ∈ $s_2$, E --> F ∈ $s_2$ , F --> C ∈ $s_2$ ⟹ a doesn't
  - b: C --> E ∈ $s_2$, D --> D ∈ $s_2$, E --> D ∈ $s_2$ , F --> E ∈ $s_2$ ⟹ b doesn't

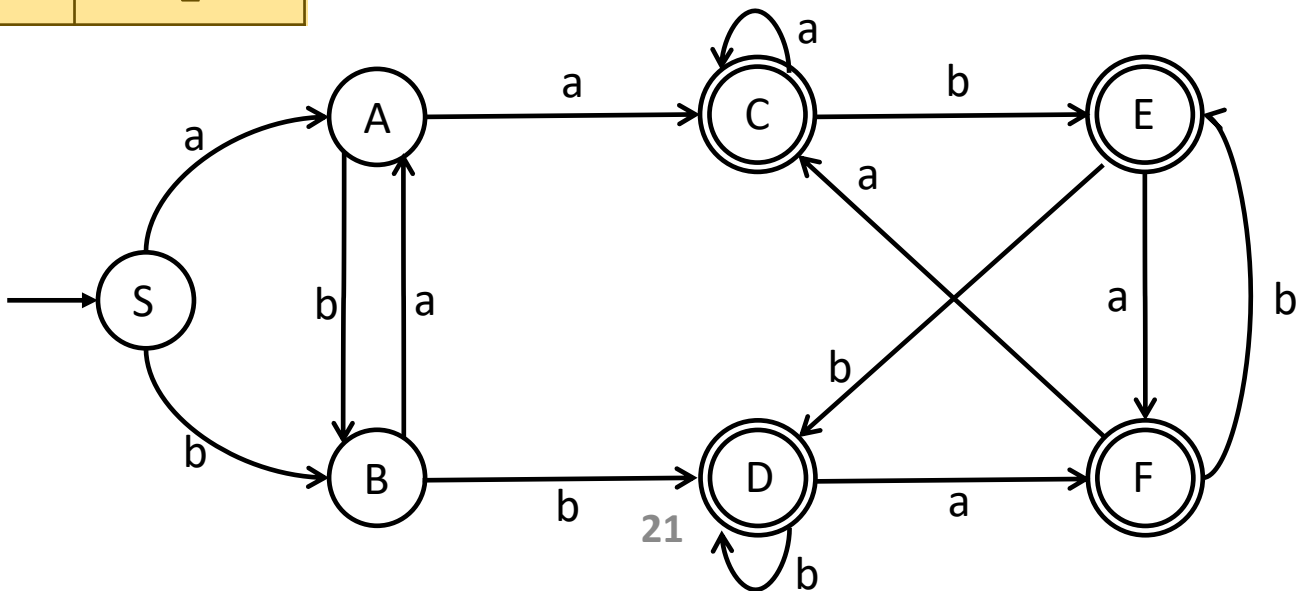


# Example (cont.)

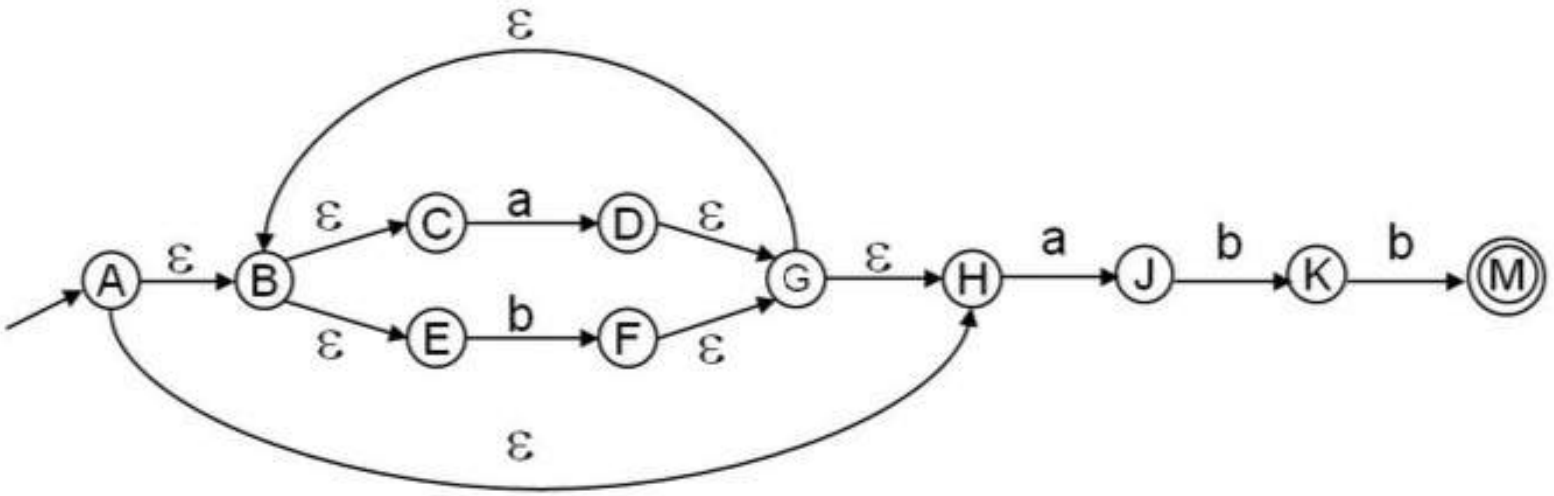
| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **C** | C | E |
| **D** | F | D |
| **E** | F | D |
| **F** | C | E |

| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **CF** | C | E |
| **DE** | F | D |

| | a | b |
|---|---|---|
| **S** | A | B |
| **A** | C | B |
| **B** | A | D |
| **CFDE** | CF | DE |



# NFA → DFA: More Example

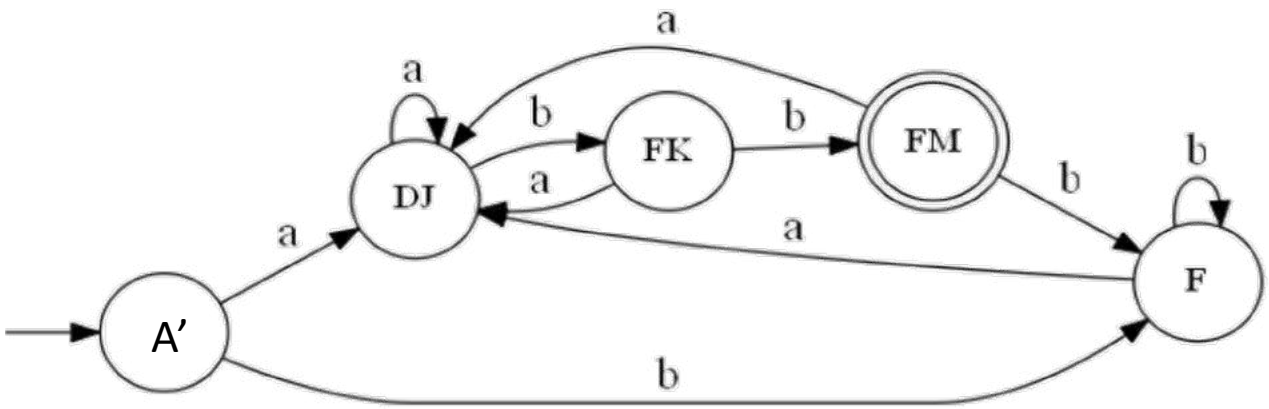



- Start state of the equivalent DFA
  - ε-closure(A) = {A, B, C, E, H} = A'
- ε-closure(move(A', a)) = ε-closure({D, J}) = {B, C, D, E, H, G, J} = B'
- ε-closure(move(A', b)) = ε-closure({F}) = {B, C, E , F, G, H} = C'
- … …

# NFA → DFA: More Example (cont.)

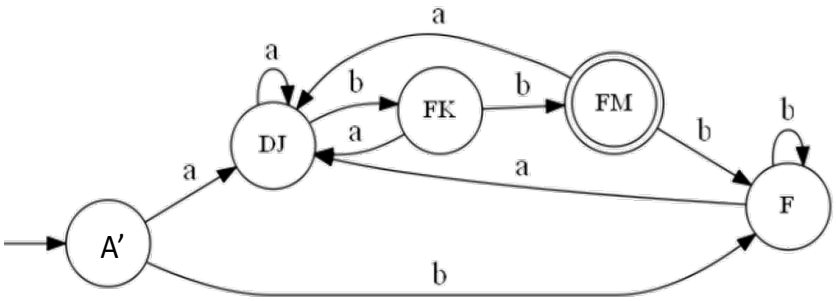
| | a | b |
|---|---|---|
| A’ | DJ | F |
| DJ | DJ | FK |
| F | DJ | F |
| FK | DJ | FM |
| FM | DJ | F |

- Is the DFA minimal?
  - States A’ and F should be merged
- Should we merge states A’ and FM?
  - NO. A’ and FM are in different sets from the very beginning (FM is accepting, A’ is not).



# NFA → DFA: More Example (cont.)

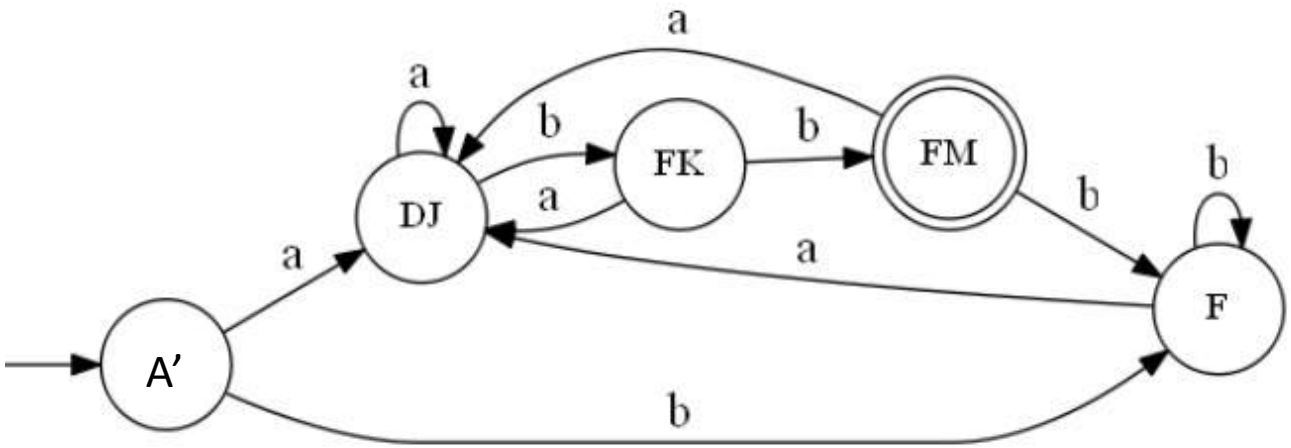
- P0: $s_1$ = {A', DJ, FK, F}, $s_2$ = {FM}
- For $s_1$, further splits into {A', DJ, F}, {FK}
  - a: A'--> DJ ∈ $s_1$, DJ --> DJ ∈ $s_1$, FK --> DJ ∈ $s_1$, F --> DJ ∈ $s_1$ ⟹ a doesn't distinct
  - b: A' --> F ∈ $s_1$, DJ --> FK ∈ $s_1$, FK --> FM ∈ $s_2$, F --> F ∈ $s_1$ ⟹ b distincts $s_1$ => $s_{11}$={A', DJ, F}, $s_{12}$={FK}
- For $s_{11}$, further splits into {A', F}, {DJ}
  - a: A'--> DJ ∈ $s_{11}$, DJ --> DJ ∈ $s_{11}$, F --> DJ ∈ $s_{11}$ ⟹ a doesn't distinct
  - b: A'--> F ∈ $s_{11}$, DJ --> FK ∈ $s_{12}$, F --> DJ ∈ $s_{11}$ ⟹ b distincts $s_{11}$ => $s_{111}$={A', F}, $s_{112}$={DJ}
- For $s_{111}$, impossible to further split
- Final states: $S_{111}$ = {A', F}, $S_{112}$ = {DJ}, $S_{12}$ = {FK}, $S_2$ = {FM}



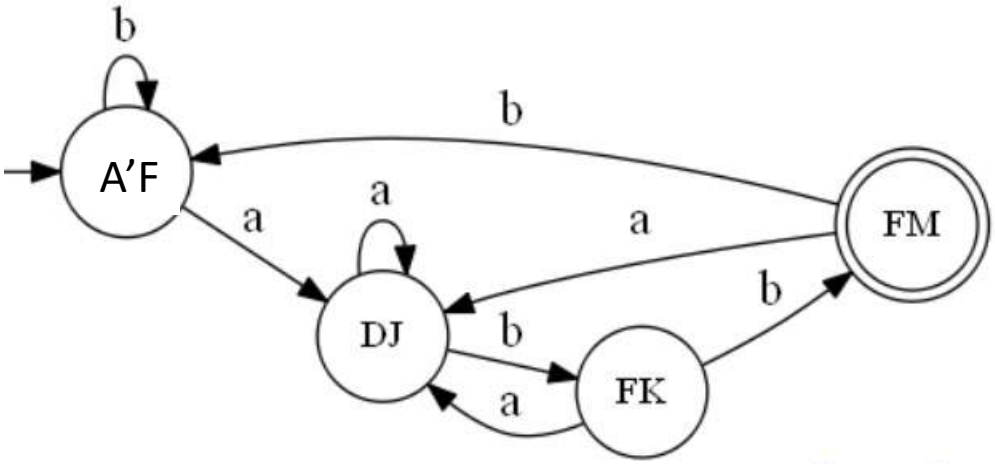

| | a | b |
|---|---|---|
| A' | DJ | F |
| DJ | DJ | FK |
| F | DJ | F |
| FK | DJ | FM |
| FM | DJ | F |

# NFA → DFA: More Example (cont.)

- Original DFA: before merging A' and F

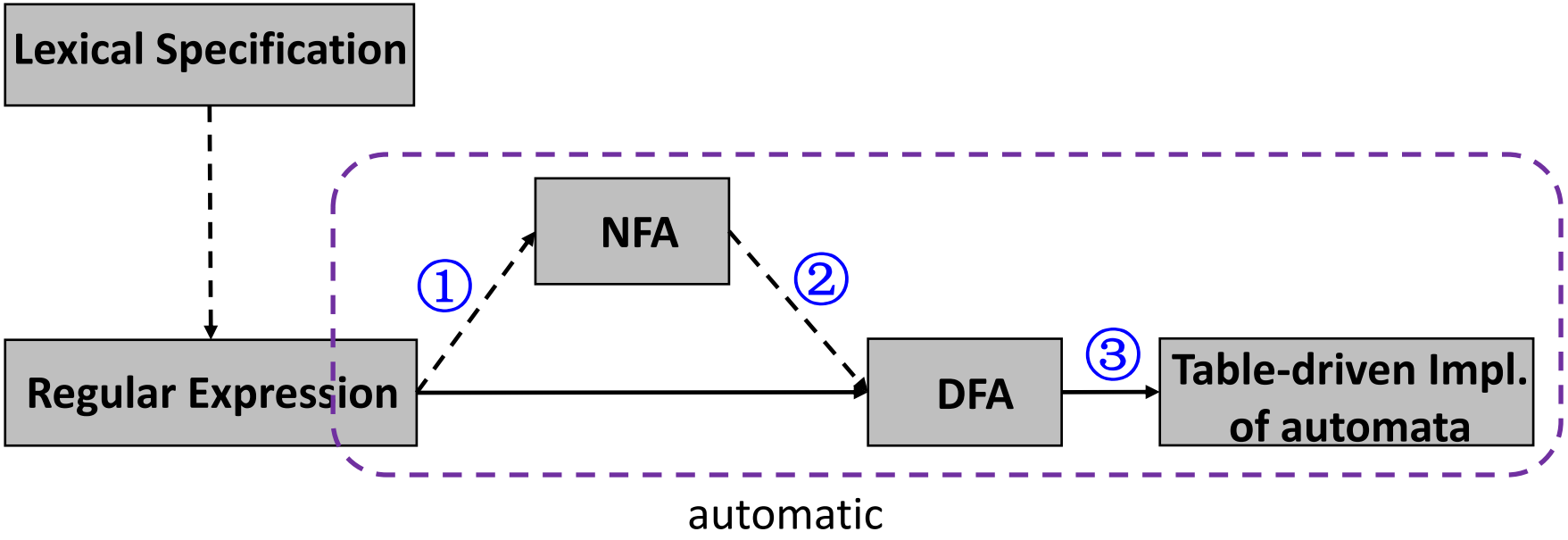



- Minimized DFA: Do you see the original RE (a|b)*abb



# The Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs (M-Y-T algorithm)
  - ② Converting NFAs to DFAs (subset construction)
  - ③' DFA minimization (partition algorithm)



# NFA → DFA: Space Complexity[空间复杂度]

- 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[时间复杂度]

- 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|)

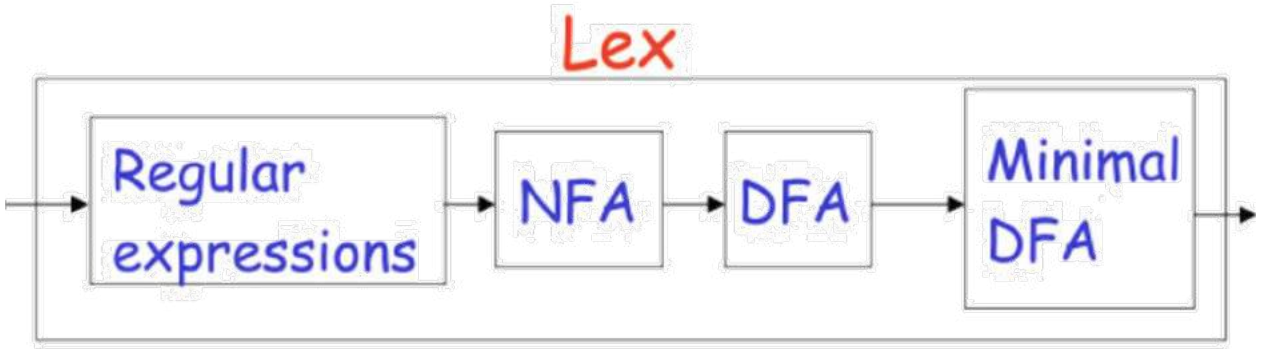
**Deterministic**: unique transition

- NFA execution
  - Requires O(|X|) steps, where |X| is the input length
    - Anyway, the input symbols should be completely processed
  - 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**: from current state, you can transit to any (including itself)

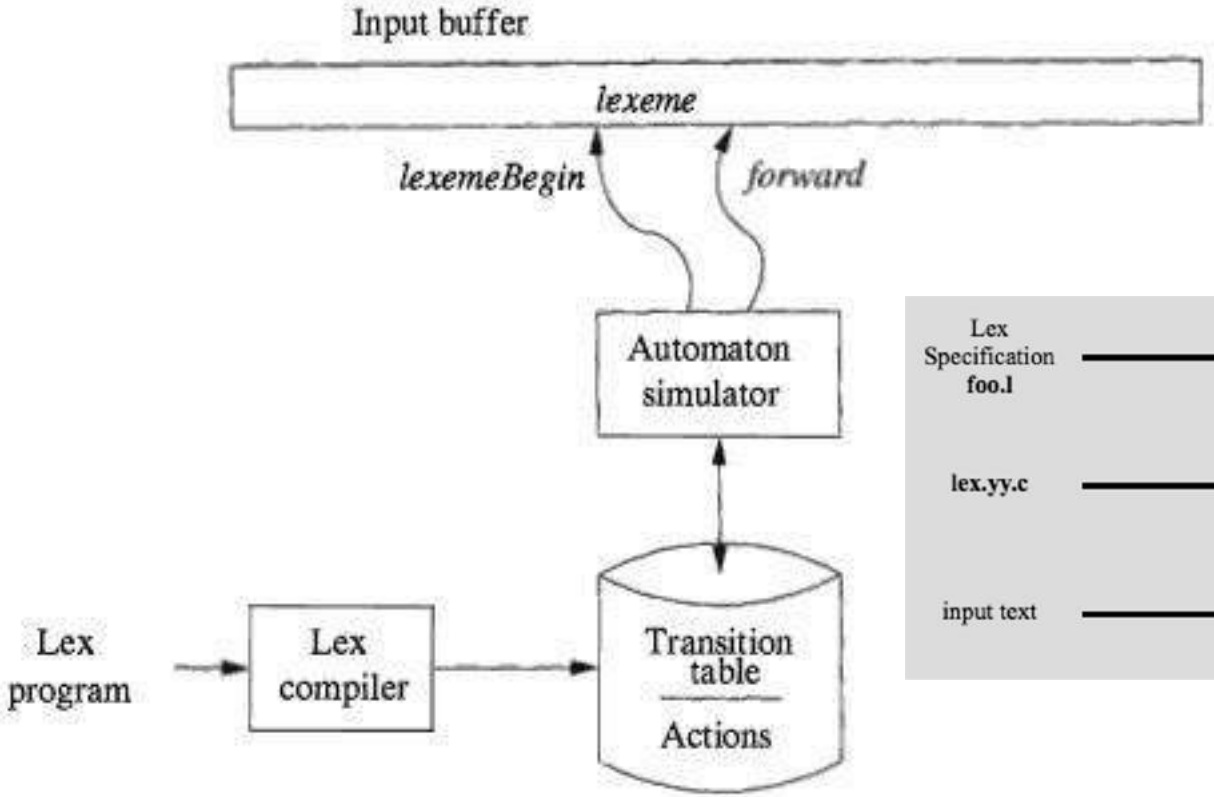
# Implementation in Practice[实际实现]

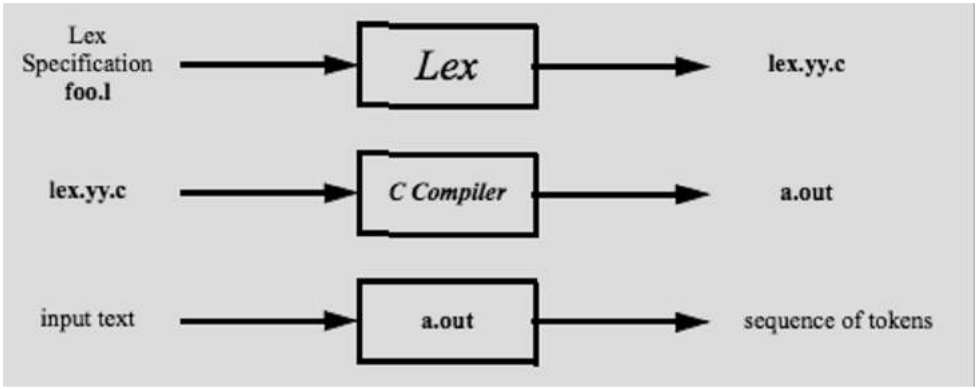
- Lex: RE → NFA → DFA → Table
  - Converts regular expressions to NFA
  - Converts NFA to DFA
  - Performs DFA state minimization to reduce space
  - Generate the transition table from DFA
  - Performs table compression to further reduce space
- Most other automated lexers also choose DFA over NFA
 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



# Lexical Analyzer Generated by Lex

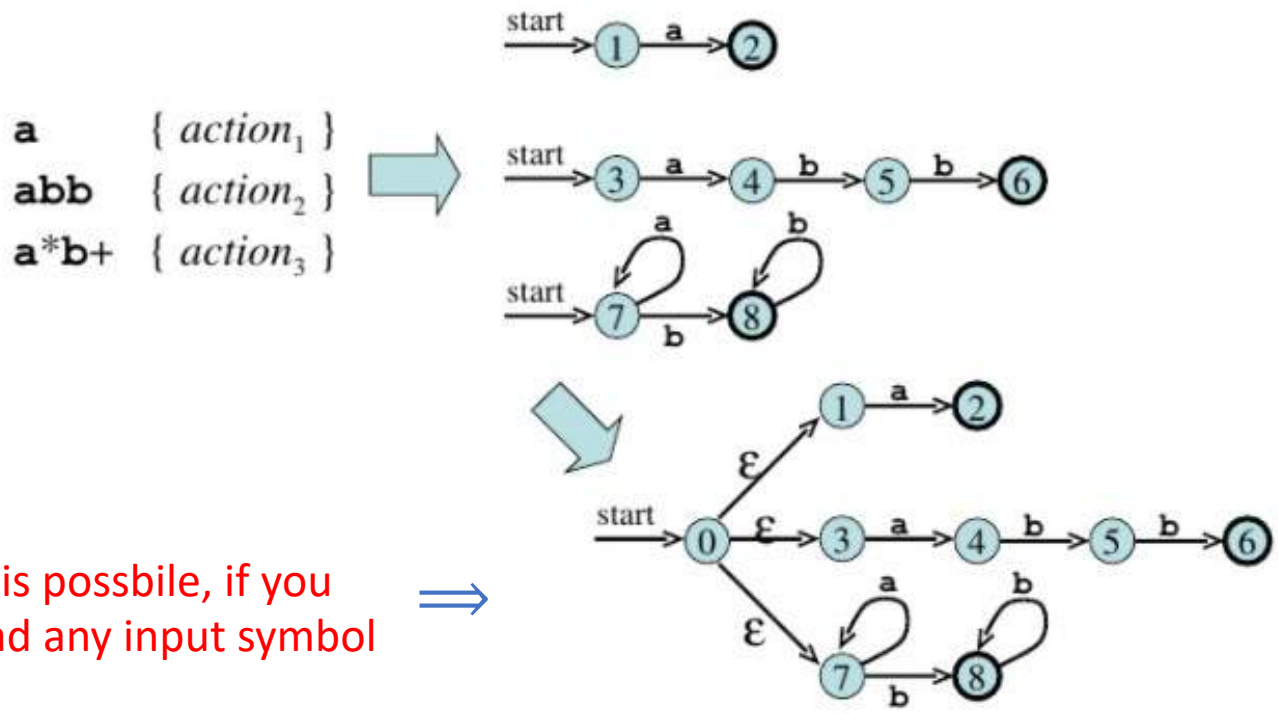
- A Lex program is turned into a transition table and actions, which are used by a FA simulator
- Automaton recognizes matching any of the patterns



# Lex: Example

- Three patterns, three NFAs
- Combine three NFAs into a single NFA
  - Add start state 0 and ε-transitions




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

# Lex: Example (cont.)

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }

%%

int main(){
  yylex();
  return 0;
}
```

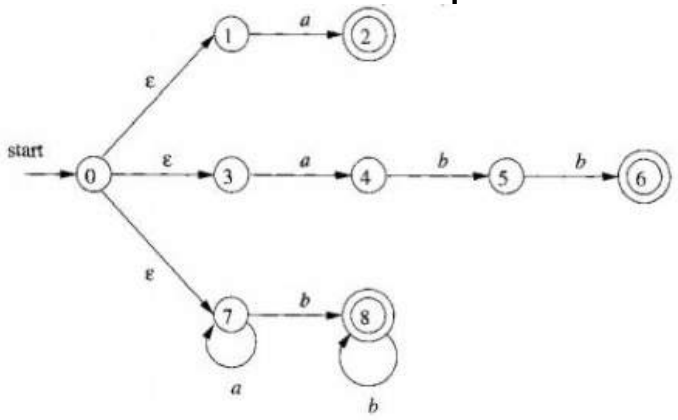
$flex lex.l
$clang lex.yy.c -o mylex -ll

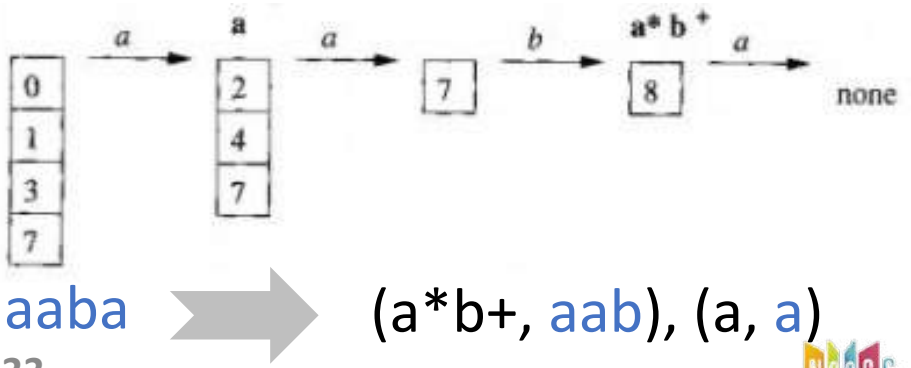
```
root@aa51dde06c76:~/test# echo "aaba" | ./mylex

<ptn3, aab>
<ptn1, a>
root@aa51dde06c76:~/test# echo "abba" | ./mylex

<ptn2, abb>
<ptn1, a>
```

# Lex: Example (cont.)

- NFA's for lexical analyzer
- 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 action$_3$
    - Return to parser indicating that token w/ pattern $p_3$=a*b+ has been found

a
abb
a*b+



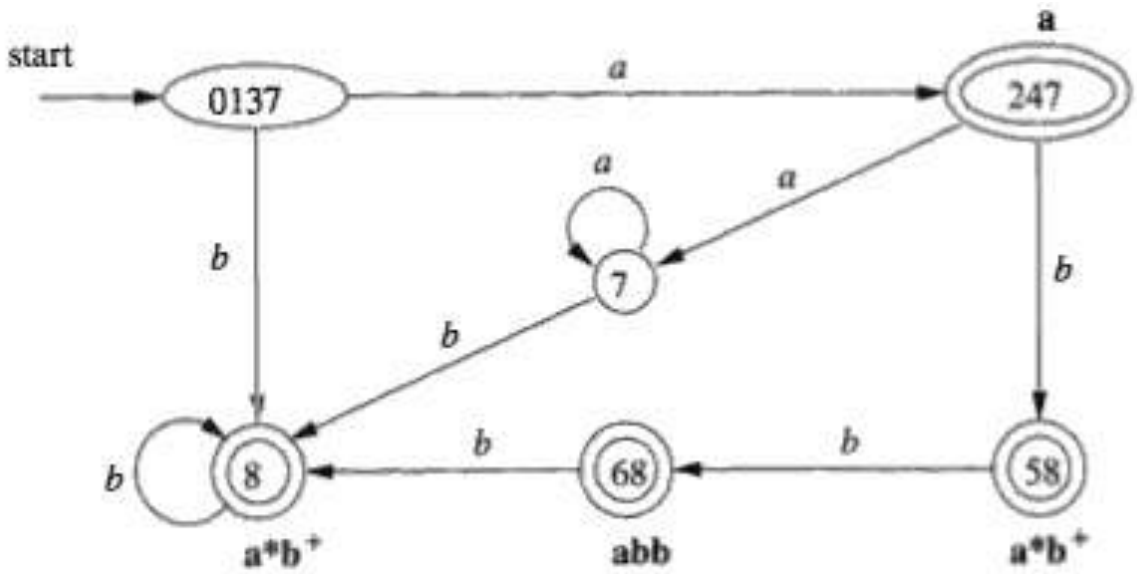




aaba ⇒ (a*b+, aab), (a, a)

# Lex: Example (cont.)

- 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 $p_2$ = abb

a
abb
a*b+



# 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 intial *a*'s followed by as many *b*'s as there are

| | |
|---|---|
| **a** | { $action_1$ } |
| **abb** | { $action_2$ } |
| **a*b+** | { $action_3$ } |

- If same length, rule appearing first takes precedence
  - String *abb* matches both the second and third
  - We consider it as a lexeme for $p_2$, since that pattern listed first

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }
```

**<ptn2, abb>**

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
```

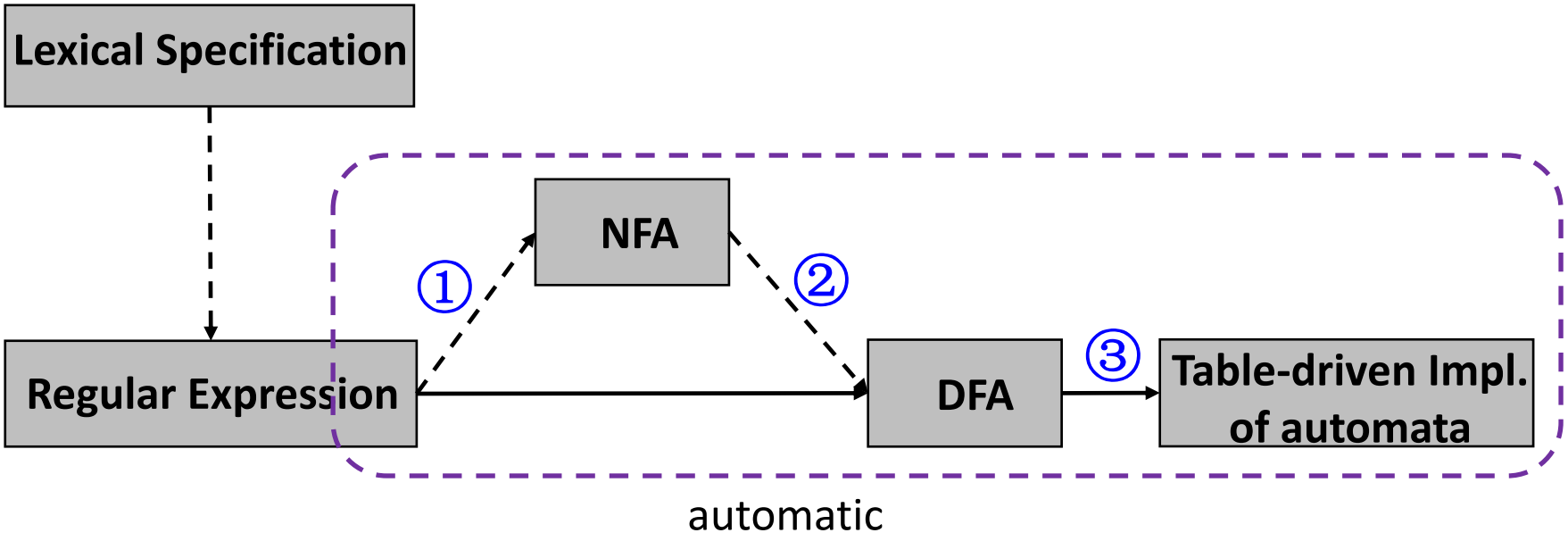
**<ptn3, abb>**

# 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 more bloated finite state machine
- **Approach 2**: recognize keywords and identifiers using 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 Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs (M-Y-T algorithm)
  - ② Converting NFAs to DFAs (subset construction)
  - ③' DFA minimization (partition algorithm)



# Beyond Regular Languages

- 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 cannot count # of times char encountered
    - $L = \{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)