Compilation Principle
编 译 原 理

第3讲：词法分析(3)

张献伟

xianweiz.github.io

DCS290, Spring 2021

Review Questions

Q1: Can we have multiple start/accepting states in FA?

start: only one, accepting: multiple

Q2: what are NFA and DFA? How to differentiate?

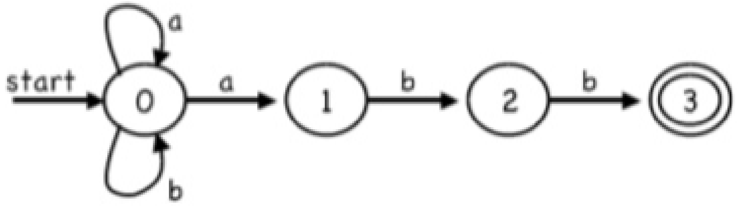
NFA: non-deterministic FA, DFA: deterministic FA
ε-move or multiple transitions per input per state

Q3: how do RE, NFA, DFA relate to each other?

L(RE) ≡ L(NFA) ≡ L(DFA)

Q4: the state graph is a NFA or DFA?
NFA, multiple transitions for state '0' on input 'a'

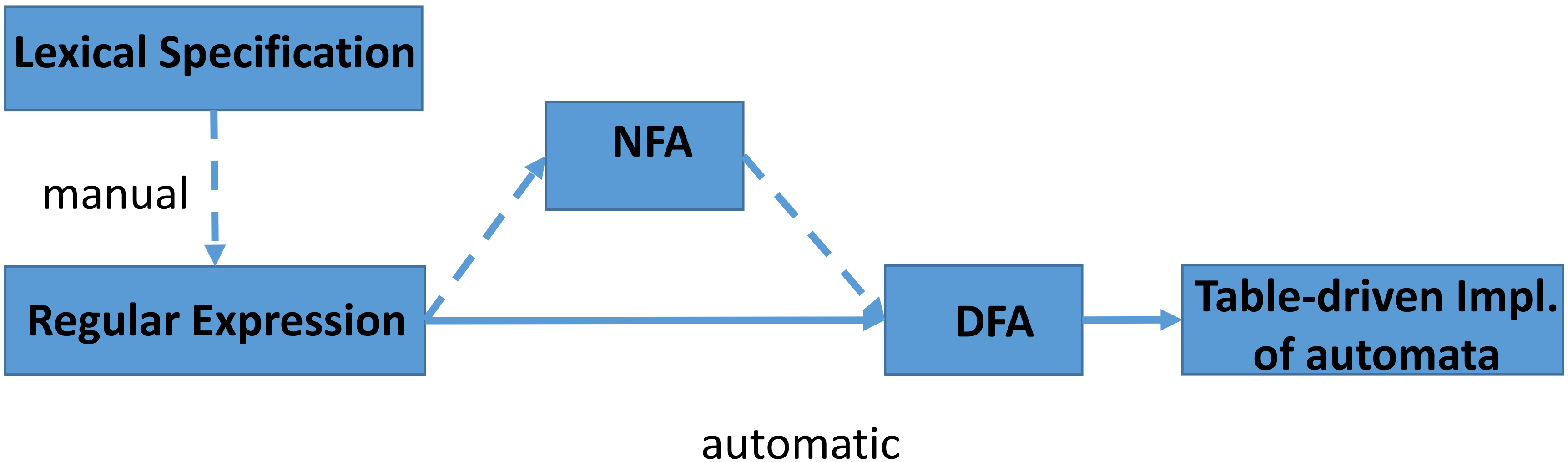
Q5: what's the language it recognizes ?



(a|b)*abb

Specification to Implementation

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



NFA → DFA: Theory

- 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)
- Resulting DFA consumes more memory than NFA
  - Potentially larger transition table
- 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 outweigh its extra memory consumption

NFA → DFA: Idea

- **Subset construction**
  - Each state of the constructed DFA corresponds to a set of NFA states
  - 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 $a_1a_2…a_n$

- Algorithm to convert
  - Input: an NFA *N*
  - Output: a DFA *D* accepting the same language as *N*

NFA → DFA: Algorithm

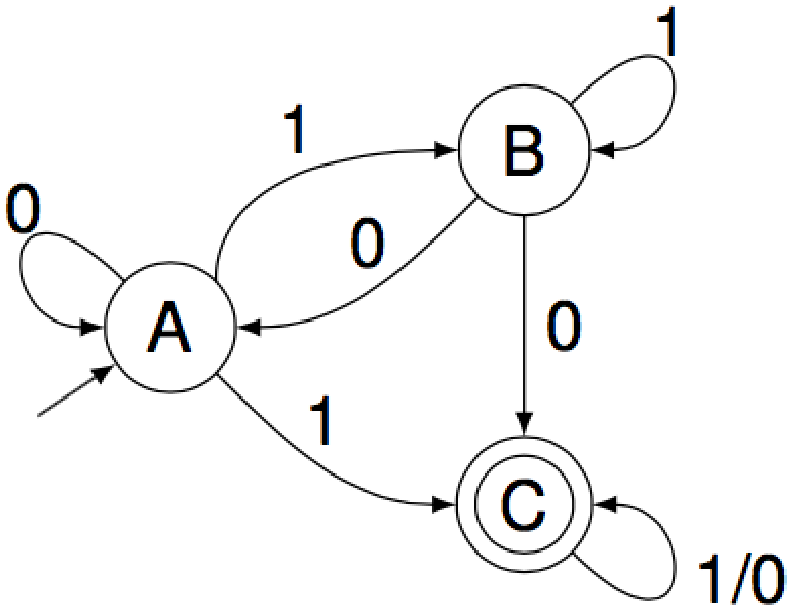
```
Initially, ε-closure(s0) is the only state in Dstates and it is unmarked
while there is an unmarked state T in Dstates do
    mark T
    for each input symbol a ∈ Σ do
        U := ε-closure(move(T,a))
        if U is not in Dstates then
            add U as an unmarked state to Dstates
        end 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***

SUN YAT-SEN UNIVERSITY

NFA → DFA: Example

- Start by constructing ε-closure of the start state
  - ε-closure(A) = A
- Keep getting ε-closure(move(T, a))
- Stop, when there are no more new states



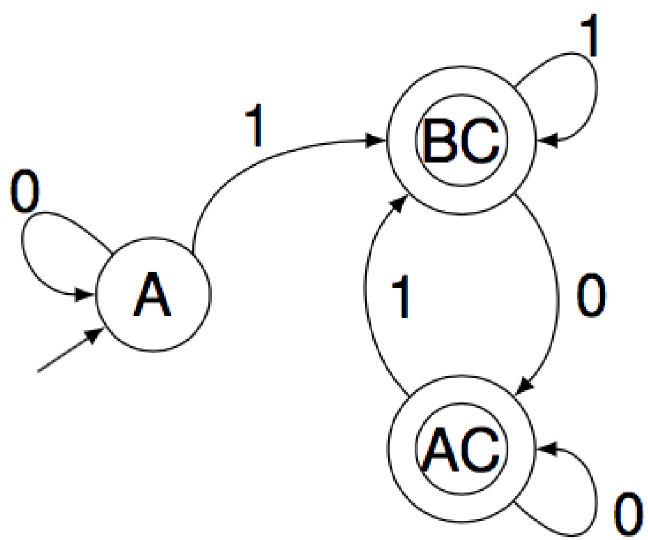
alphabet →

state ↓

	0	1
A	**A**	**BC**
BC	**AC**	**BC**
AC	**AC**	**BC**

NFA → DFA: Example (cont.)

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



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

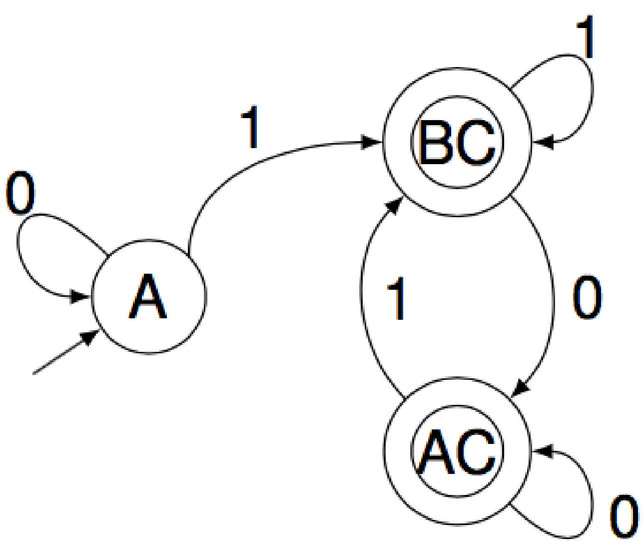
alphabet →

state ↓

	0	1
A	**A**	**BC**
BC	**AC**	**BC**
AC	**AC**	**BC**

NFA → DFA: Minimization

- Any DFA can be converted to its minimum-state equivalent DFA
  - Partitioning the states of a DFA into groups of states that cannot be distinguished
  - Each groups of states is then merged into a single state of the min-state DFA



Initial: {A}, {BC, AC}

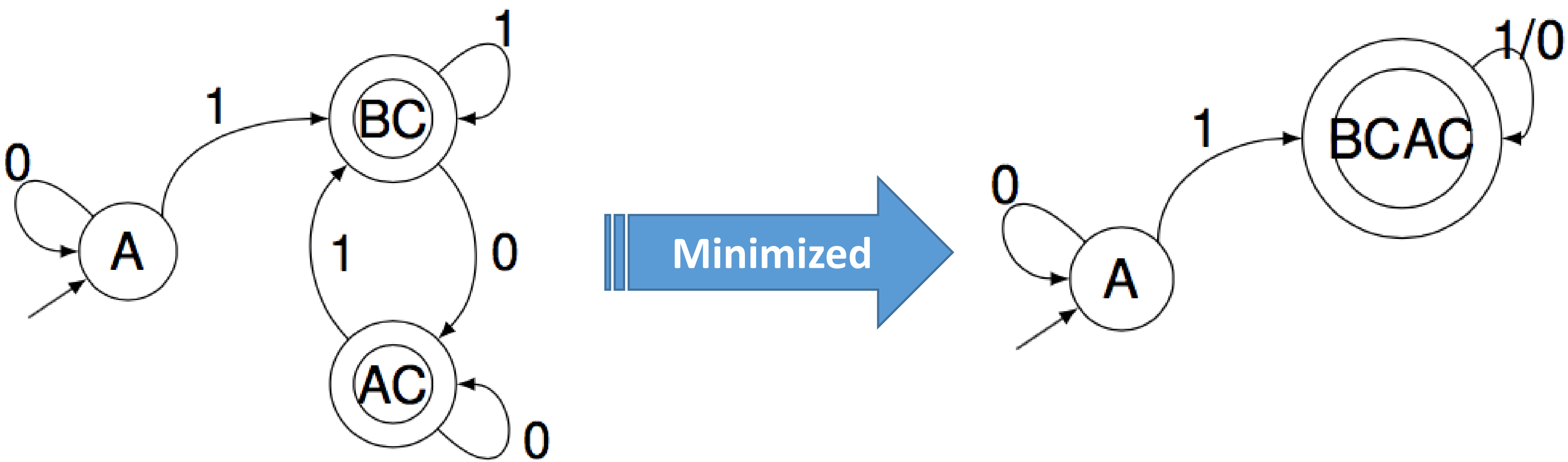
For {BC, AC}

- 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’

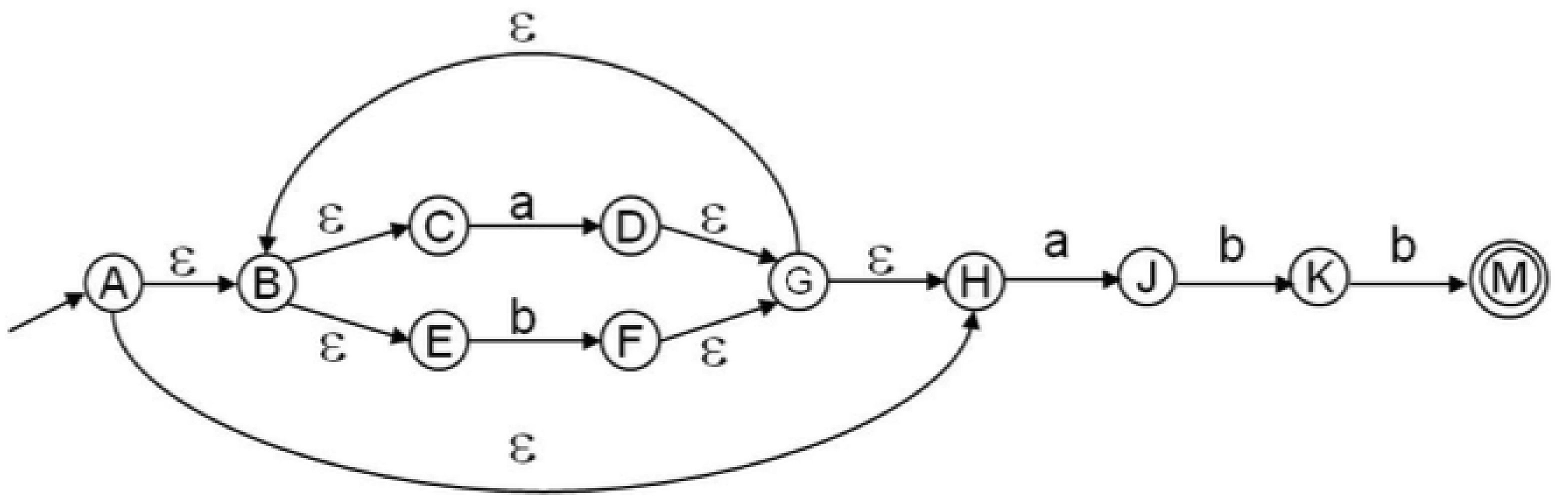
Final: {A}, {BCAC}

NFA → DFA: Minimization (cont.)

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



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: Space Complexity

- 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

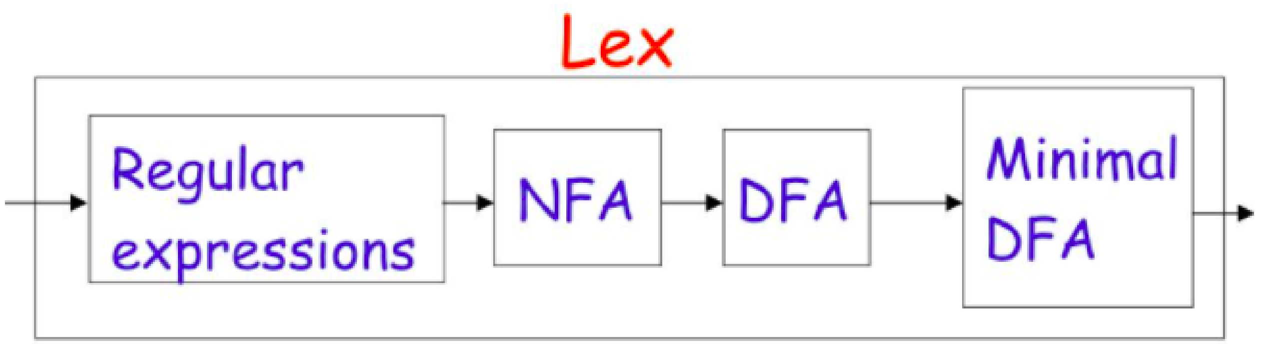
- The resulting DFA has $O(2^N)$ space complexity, where N is number of original states in NFA
  - For real languages, the NFA and DFA have about same #states

NFA → DFA: Time Complexity

- 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 = $O(|X|*N^2)$

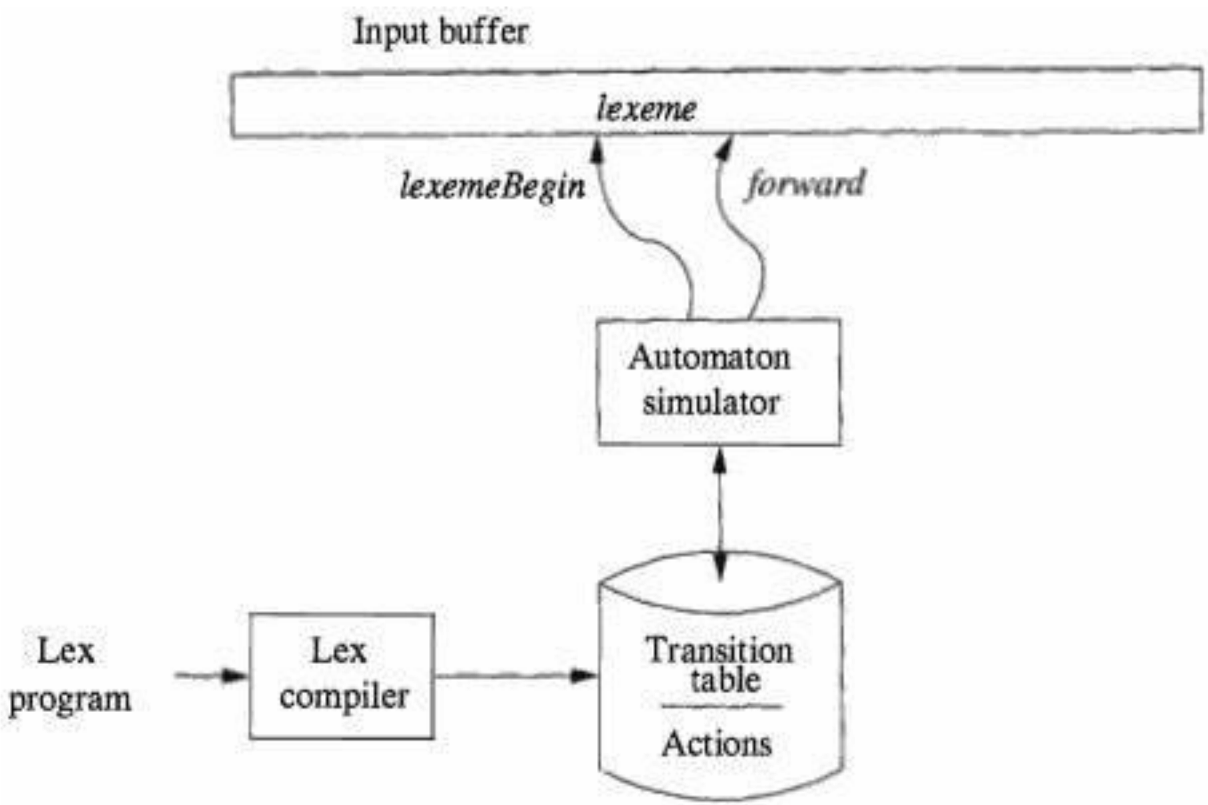
Implementation in Practice

- 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



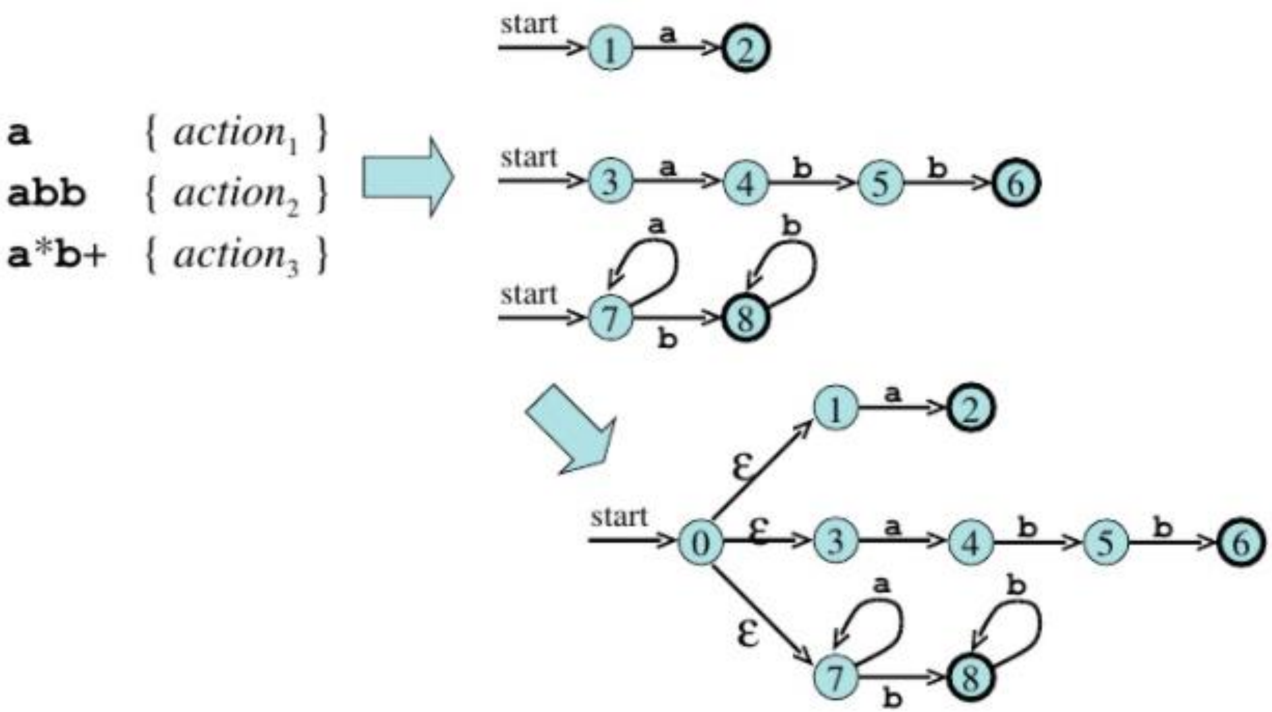
Lexical Analyzer Generated by Lex

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



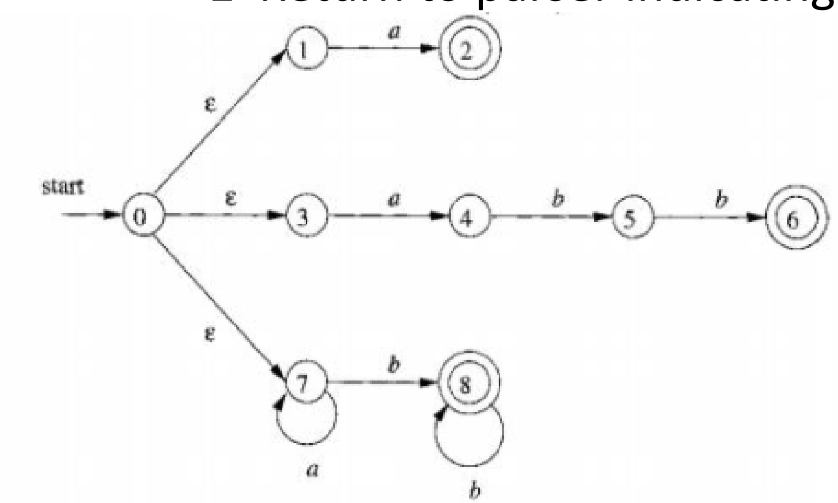
Lex: Example

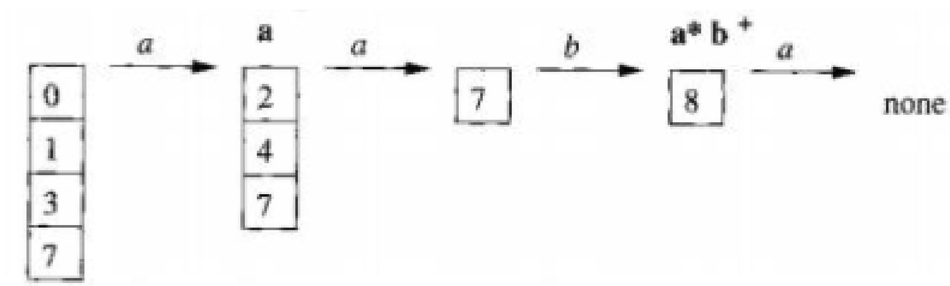
- Three patterns, three NFAs
- Combine three NFAs into a single NFA
  - Add start state 0 and ε-transitions



Lex: Example (cont.)

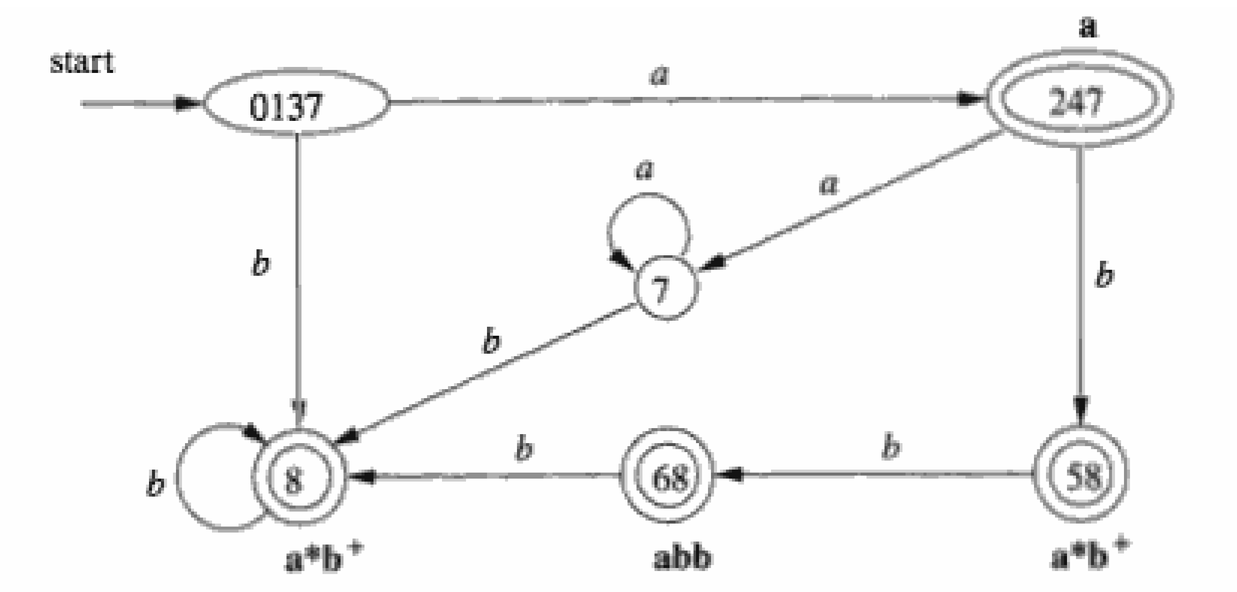
- NFA's for lexical analyzer
- Input: aaba
  - ε-closure(0) = {0, 1, 3, 7}
  - Empty states after reading the fourth input symbol
    - There are no transitions out of state 8
    - Back up, looking for a set of states that include an accepting state
  - State 8: a*b+ has been matched
    - Select aab as the lexeme, execute action A_3
    - Return to parser indicating that token w/ pattern p_3=a*b+ has been found





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



How Much Should We Match?

- 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
- 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

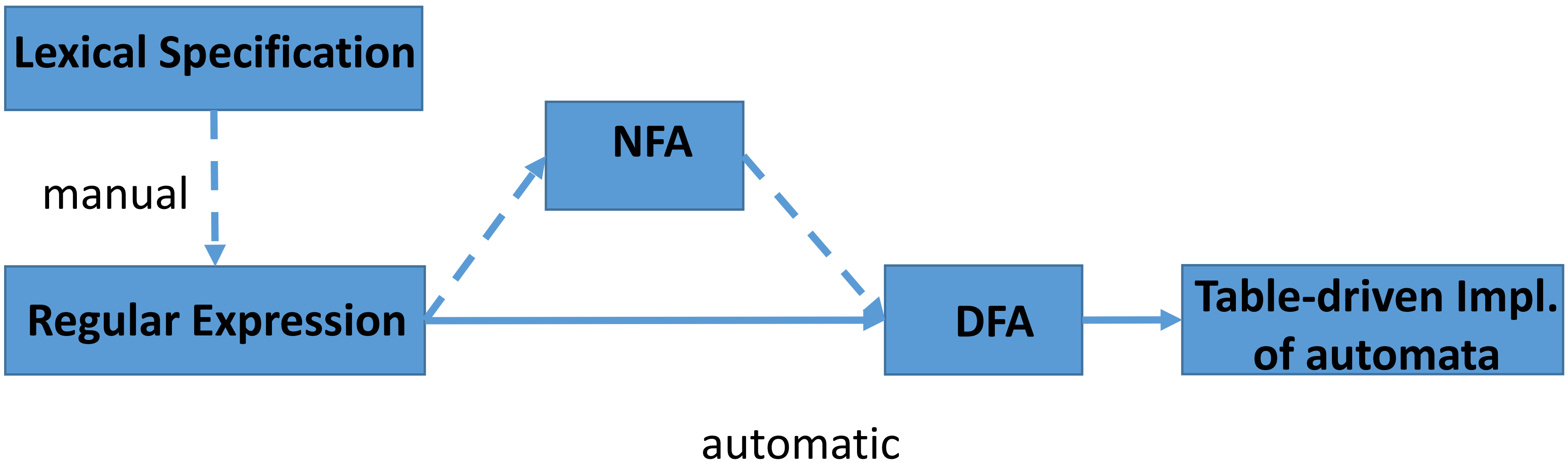
a	{ $action_1$ }
abb	{ $action_2$ }
a*b+	{ $action_3$ }

How to Match Keywords?

- 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

Conversion Flow

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



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
  - Finite automata 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
  - Next, we will discuss context-free languages (CFGs)