Implementation of Lexical Analysis

CS143

Lecture 4

Tips on Building Large Systems

- KISS (Keep It Simple, Stupid!)
- Don’t optimize prematurely
- Design systems that can be tested
- It is easier to modify a working system than to get a system working

Outline

- Specifying lexical structure using regular expressions

- Finite automata
  – Deterministic Finite Automata (DFAs)
  – Non-deterministic Finite Automata (NFAs)

- Implementation of regular expressions

Notation

- There is variation in regular expression notation
  - At least one A: $A^+ \equiv AA^*$
  - Union: $A + B \equiv A \mid B$
  - Option: $A + \varepsilon \equiv A?$
  - Range: 'a'+'b'+…+'z' ≡ [a-z]
  - Excluded range:
    complement of [a-z] ≡ [^a-z]

Lexical Specification → Regex in five steps

1. Write a regex for each token
   - Number = digit +
   - Keyword = ‘if’ + ‘else’ + …
   - Identifier = letter (letter + digit)*
   - OpenPar = ‘(‘
   - …

Lexical Specification → Regex in five steps

2. Construct R, matching all lexemes for all tokens

$$R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots$$
$$= R_1 + R_2 + \ldots$$

(This step is done automatically by tools like flex)

Lexical Specification → Regex in five steps

3. Let input be $x_1 \ldots x_n$

   For $1 \leq i \leq n$ check

   $$x_1 \ldots x_i \in L(R)$$

4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j

5. Take $x_1 \ldots x_i$ as token, and go to (3) for next token.

Ambiguity 1

- There are ambiguities in the algorithm

- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$

- Rule: Pick longest possible string in $L(R)$
  - Pick k if $k > i$
  - The “maximal munch”

Ambiguity 2

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$

- Rule: use rule listed first
  - Pick j if $j < k$
  - E.g., treat "if" as a keyword, not an identifier

Error Handling

- What if

  No rule matches a prefix of input ?
- Problem: Can’t just get stuck …
- Solution:

Error Handling

- What if

  No rule matches a prefix of input ?
- Problem: Can’t just get stuck …
- Solution:

  – Write a rule or regex for matching all “bad” strings
  – Put it last (lowest priority)

Summary

- Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve
  - To handle errors

- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

Summary

- Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
    sol: [matches longest possible and highest priority]
  - To handle errors

- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

Summary

- Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
    sol: [matches longest possible and highest priority]
  - To handle errors
    sol: [define a regex for all erroneous or bad string]

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A finite set of states S
  - A start state n
  - A set of accepting(final) states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

- Is read

In state s_1 on input "a" go to state s_2

Finite Automata

- Transition

$$s_1 \to^a s_2$$

- Is read

In state s_1 on input "a" go to state s_2

- If end of input and in accepting state => accept

Finite Automata

- Transition

$$s_1 \to^a s_2$$

- Is read

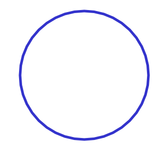
In state s_1 on input "a" go to state s_2

- If end of input and in accepting state => accept

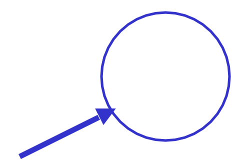
- Otherwise => reject

Finite Automata State Graphs

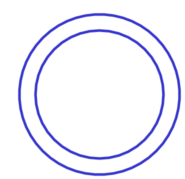
- A state



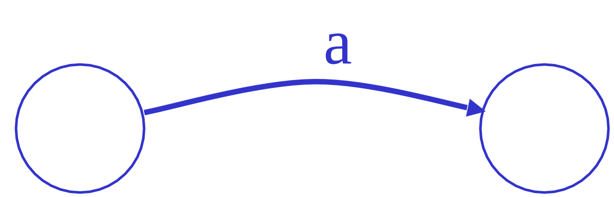
- The start state



- An accepting state

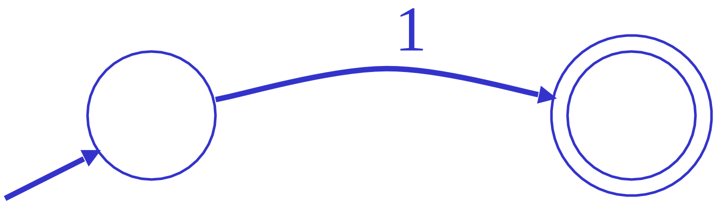


- A transition



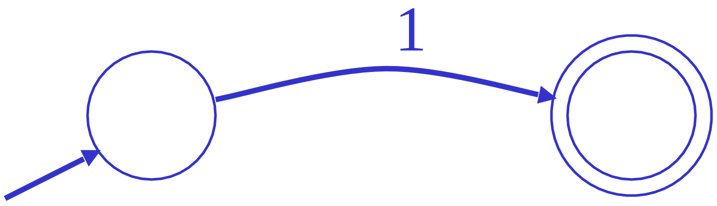
A Simple Example

- A finite automaton that accepts only "1"



A Simple Example

- A finite automaton that accepts only “1”

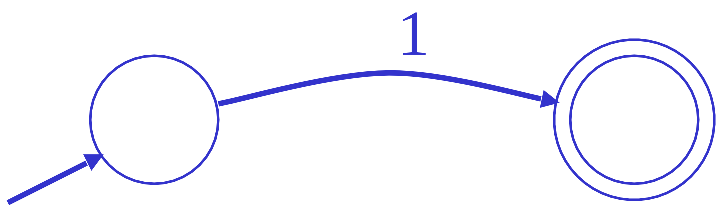


Accept

State	Input
A	^1
B	1^

A Simple Example

- A finite automaton that accepts only "1"

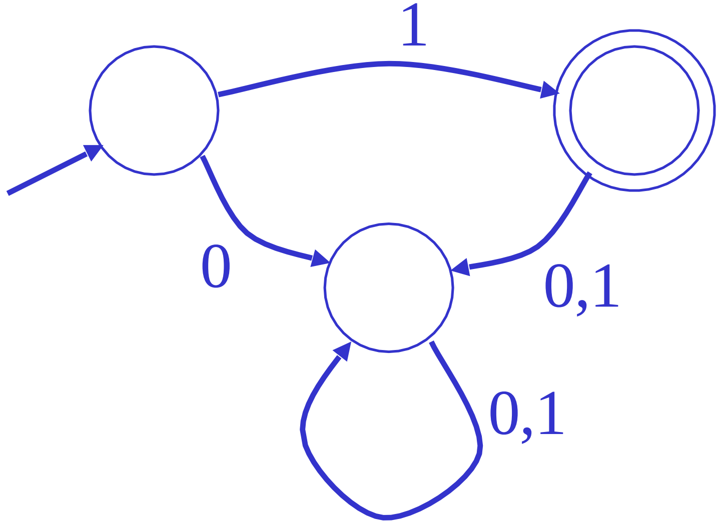


Reject

State	Input
A	^0

A Simple Example

- A finite automaton that accepts only "1"

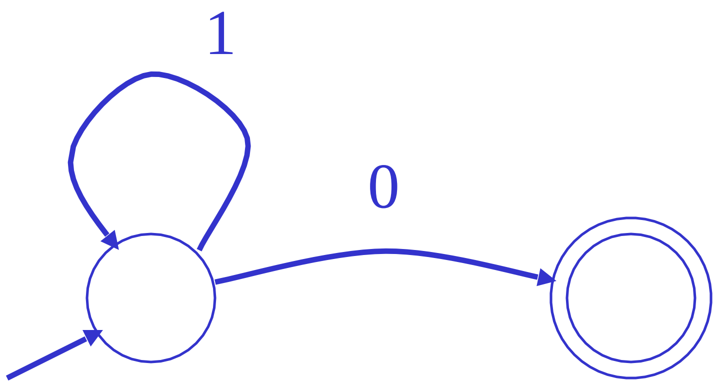


Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

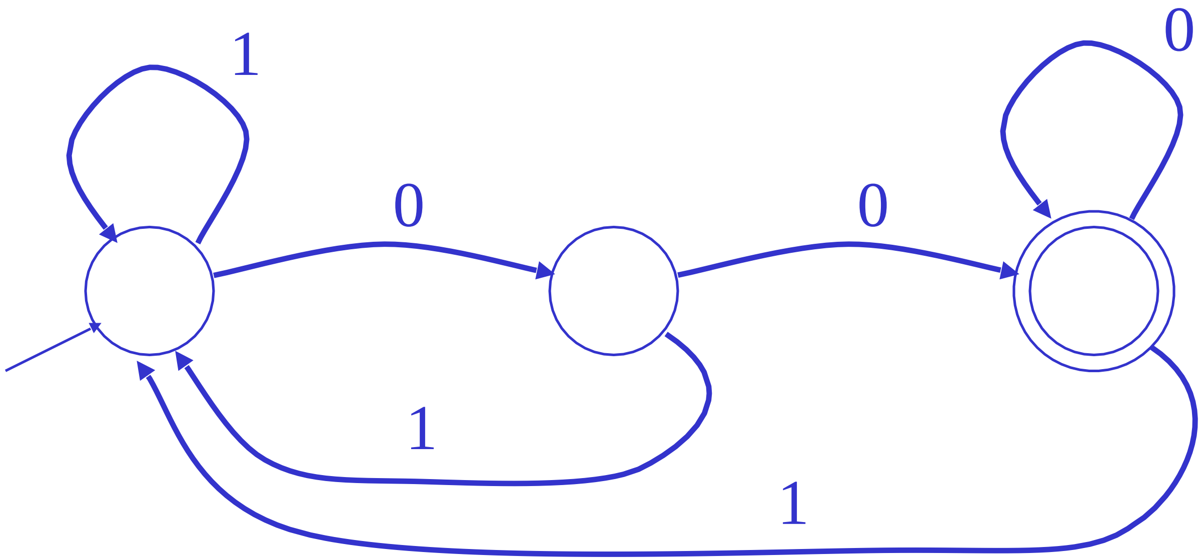
Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



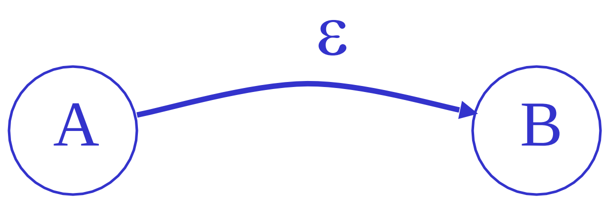
And Another Example

- Alphabet {0,1}
- What language does this recognize?



Epsilon Moves in NFAs

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

- Only exist in NFAs

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - Exactly one transition per input per state
  - No ε-moves

- Nondeterministic Finite Automata (NFA)
  - Can have zero, one, or multiple transitions for one input in a given state
  - Can have ε-moves

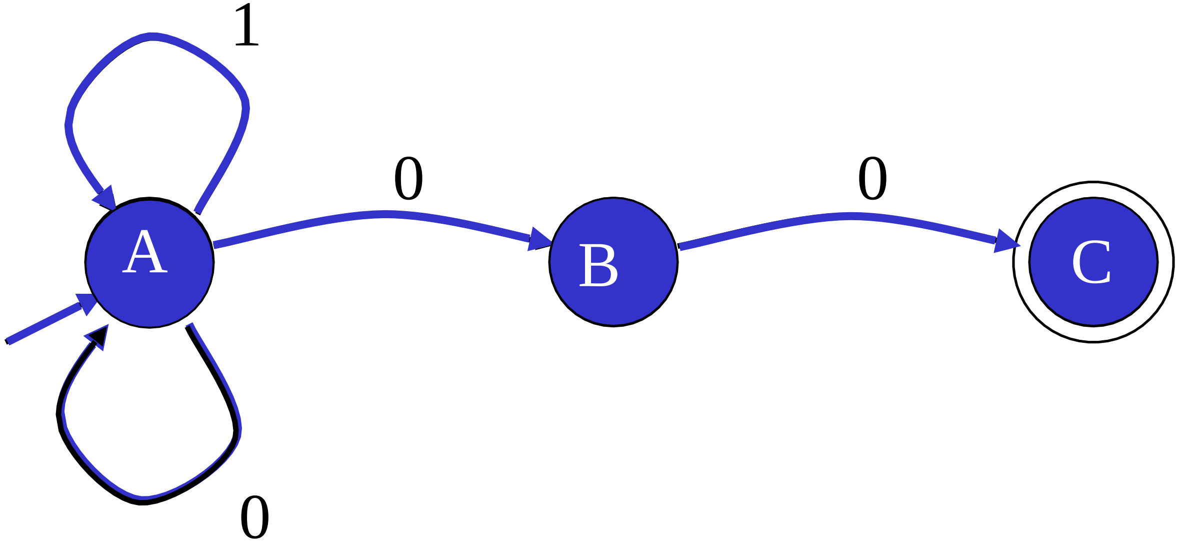
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states

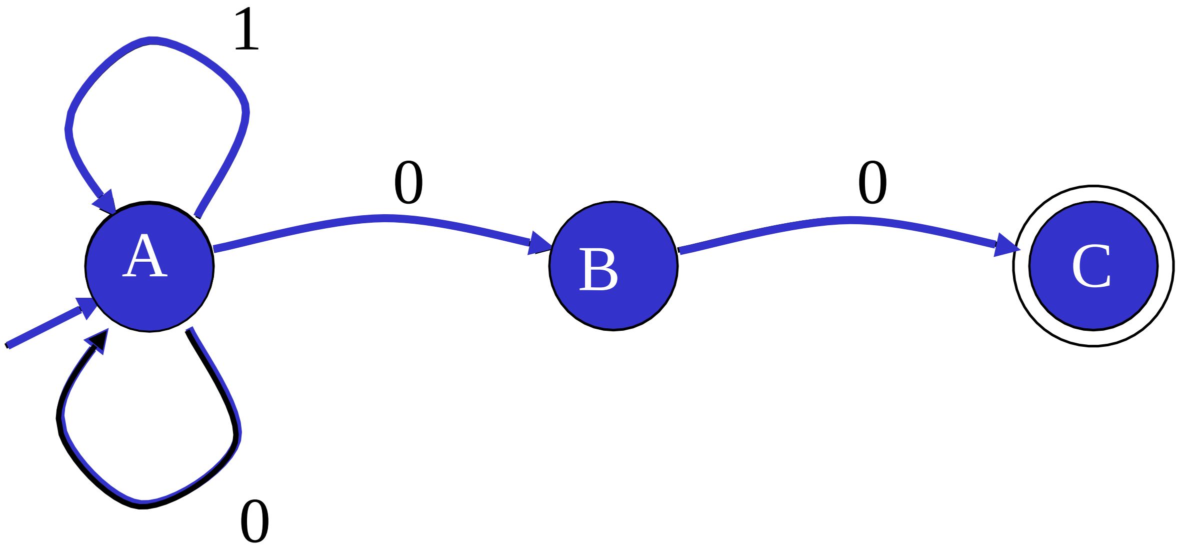


- Input: 1 0 0
- State

Rule: NFA accepts if it can get to a final state

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 0
- State: {A} {A,B} {A,B,C}

Rule: NFA accepts if it <u>can</u> get to a final state

NFA vs. DFA (1)

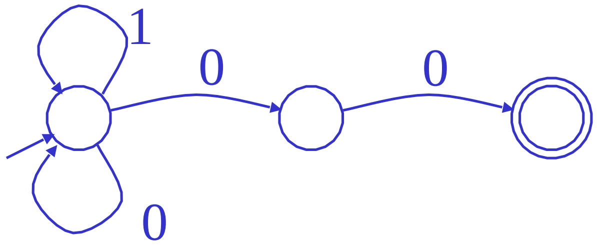
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute
  - There are no choices to consider

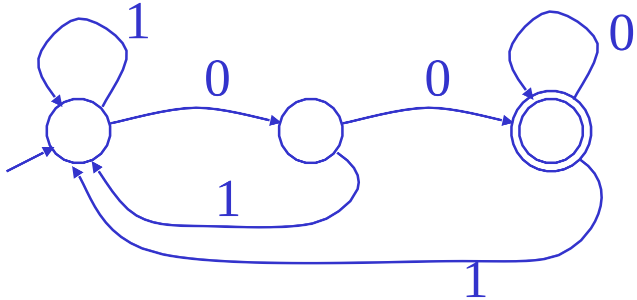
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



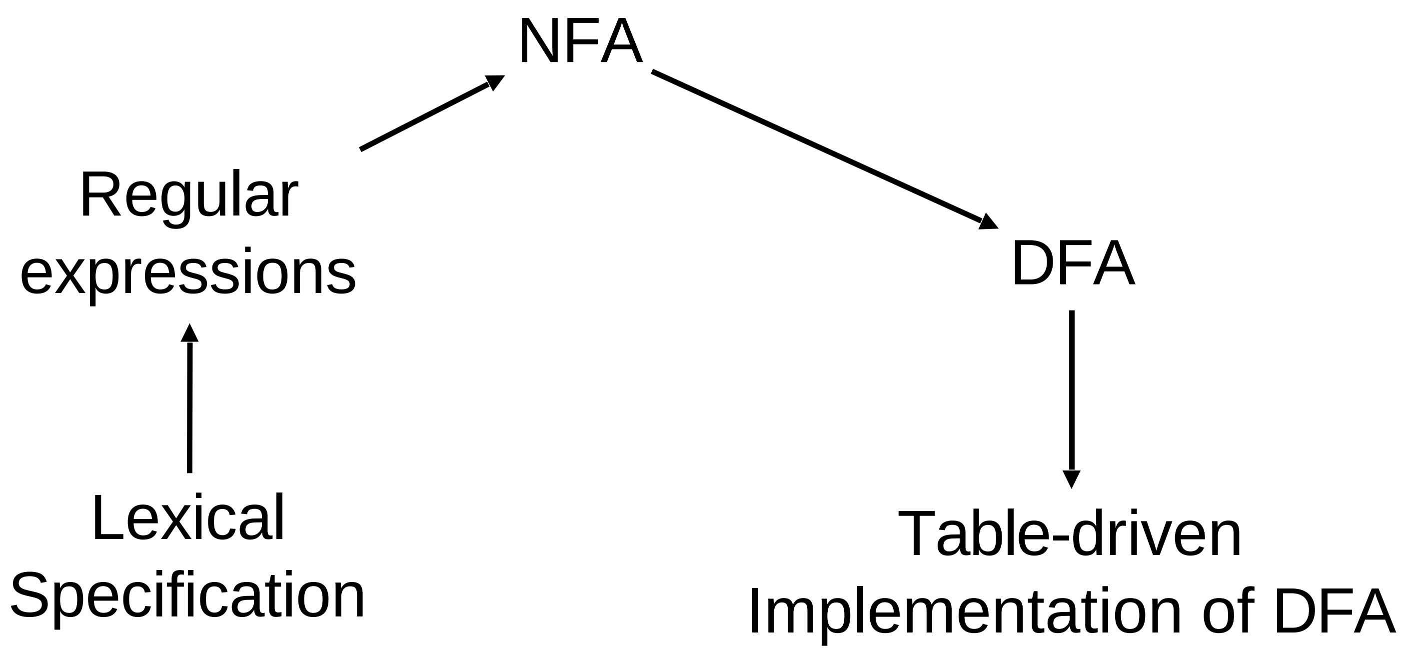
DFA



- DFA can be exponentially larger than NFA

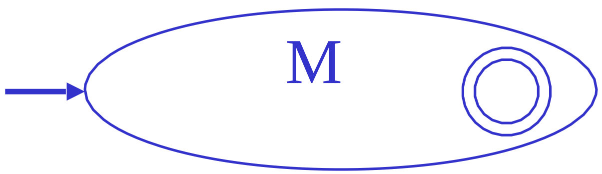
Convert Regular Expressions to Finite Automata

- High-level sketch

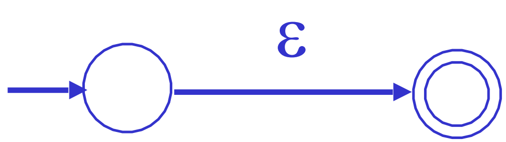


Convert Regular Expressions to NFA (1)

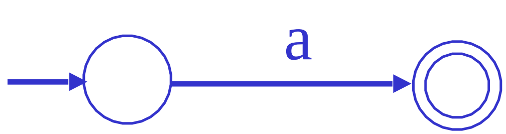
- For each kind of regex, define an equivalent NFA
  - Notation: NFA for regex M



- For ε

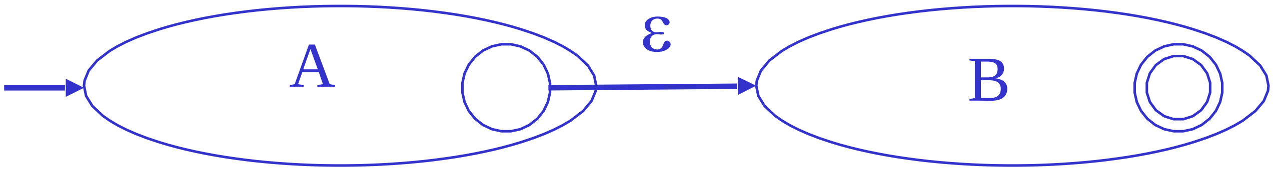


- For input a

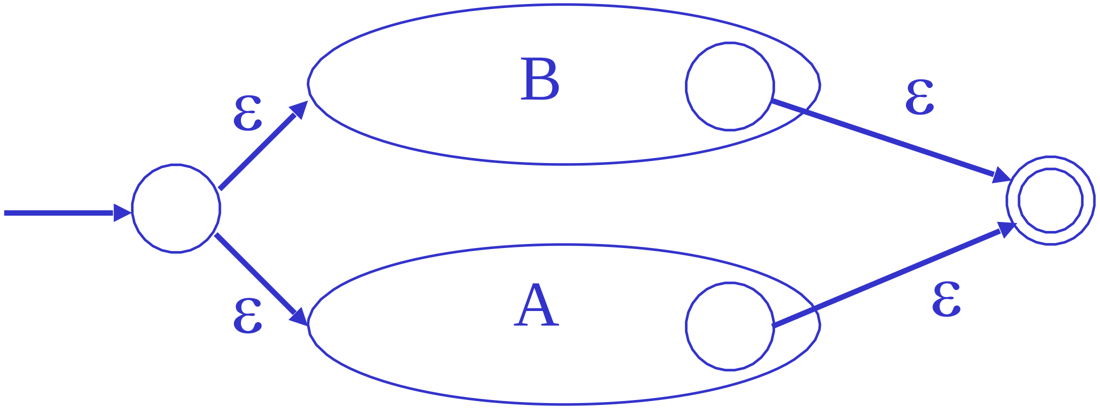


Convert Regular Expressions to NFA (2)

- For AB

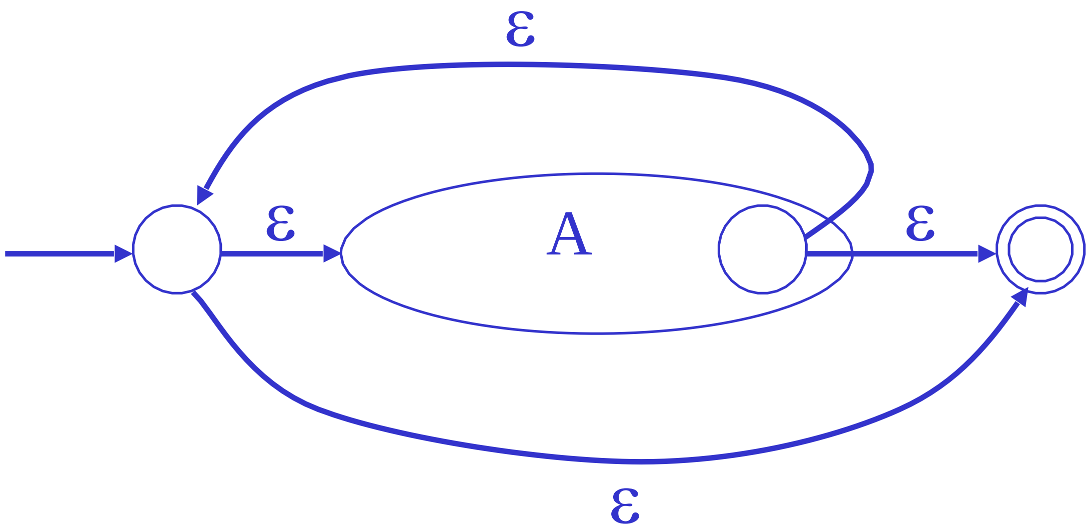


- For A + B



Convert Regular Expressions to NFA (3)

- For A*

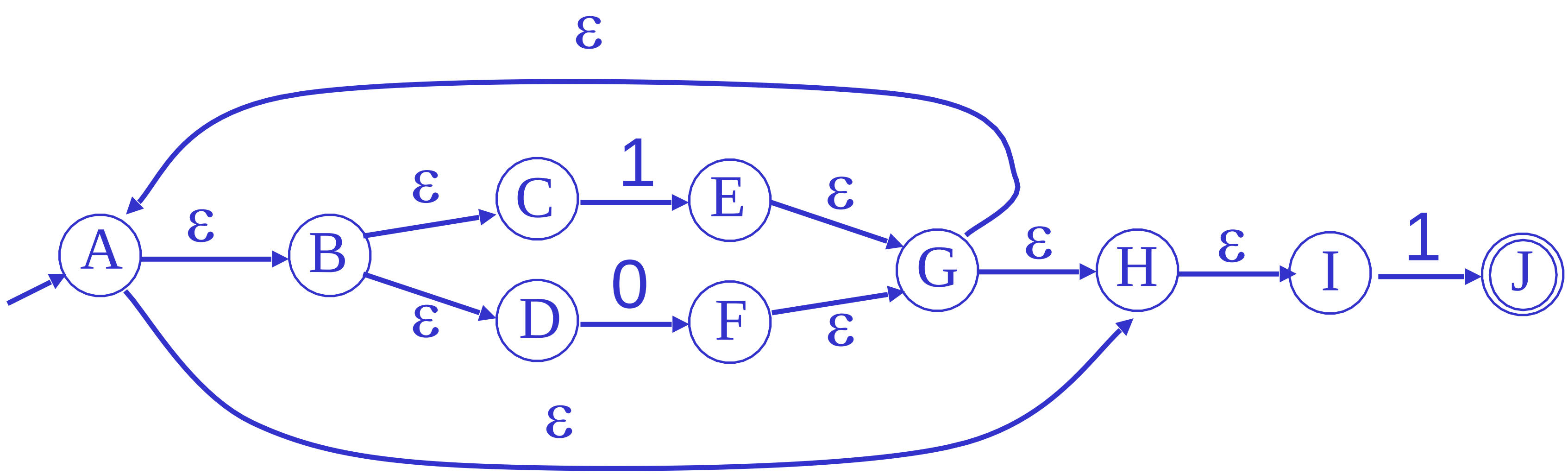


Example of RegExp to NFA conversion

- Consider the regular expression

  $(1+0)^*1$

- The NFA is



NFA to DFA. Remark

- An NFA may be in many states at any time

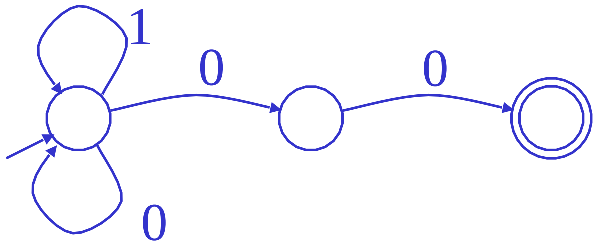
How many different states ?

- If there are N states, the NFA must be in some subset of those N states

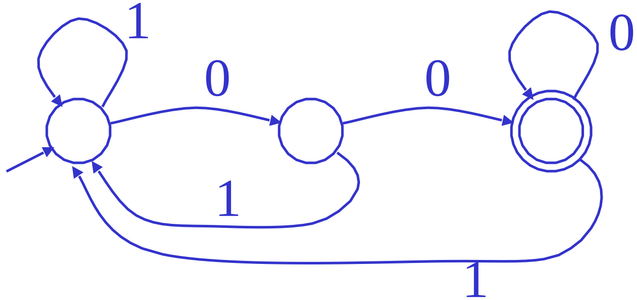
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



DFA



Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

input symbols

states	0	1
a	**a**	**b**
b	a	b
c	b	b
d	a	b

Table Implementation of a DFA

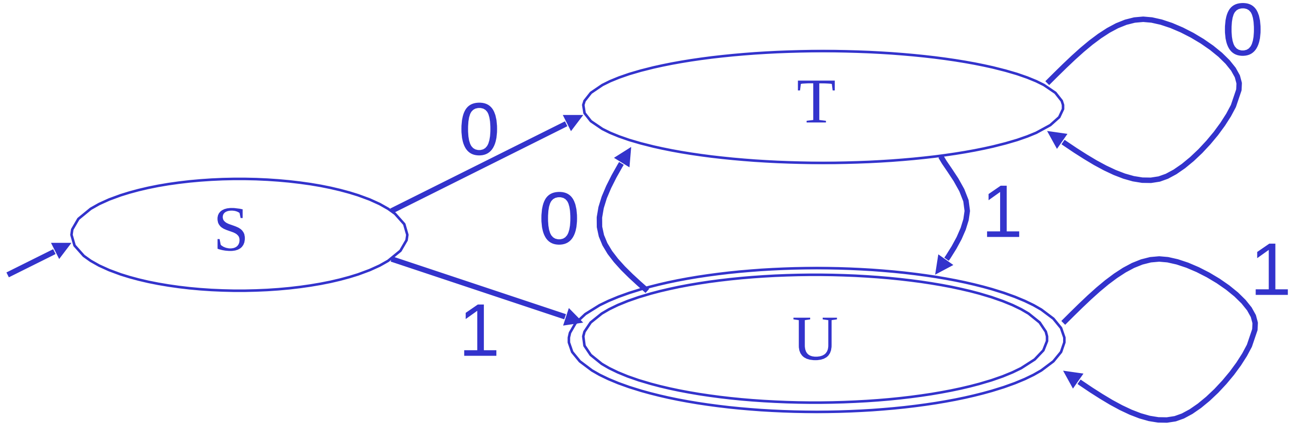
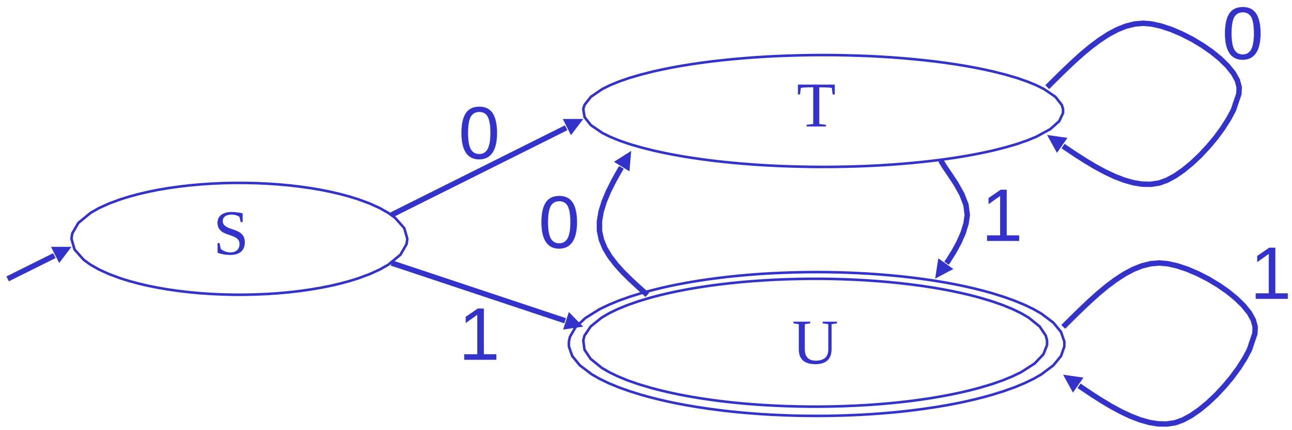


Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex

- But, DFAs can be huge

- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations