Lexical Analysis part II

Chapter 3

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

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

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

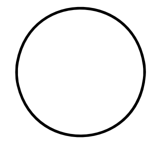
- Is read

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

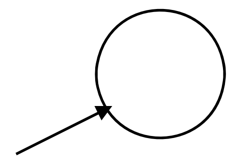
- If end of input (or no transition possible)
  - If in accepting state => accept
  - Otherwise => reject

Finite Automata State Graphs

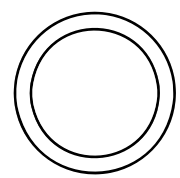
- A state



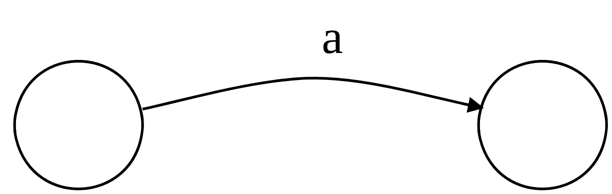
- The start state



- An accepting state



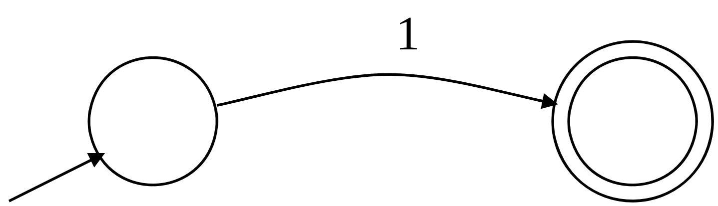
- A transition



A Simple Example

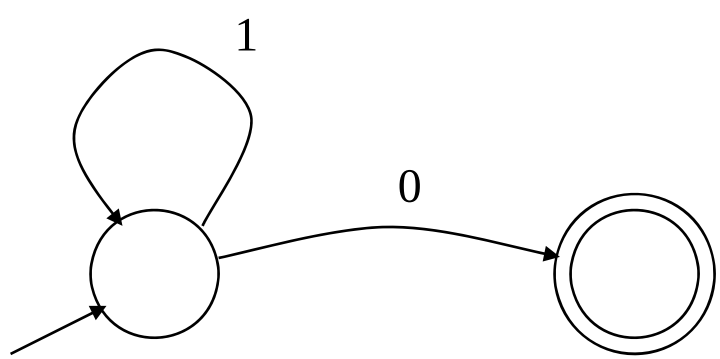
- A finite automaton that accepts only “1”

- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state



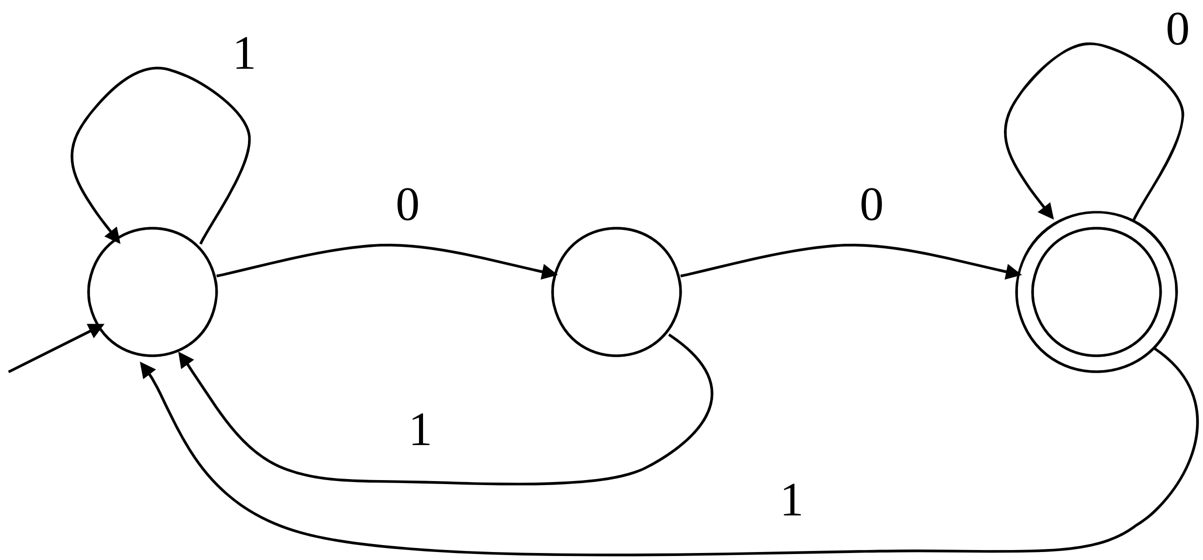
Another Simple Example

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



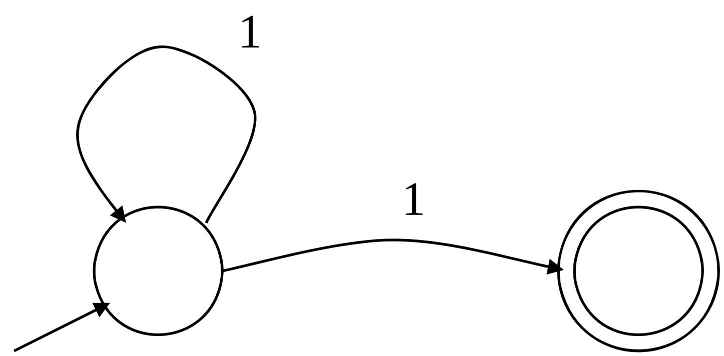
And Another Example

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



And Another Example

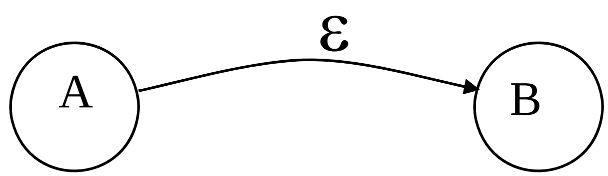
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



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

Deterministic and Nondeterministic Automata

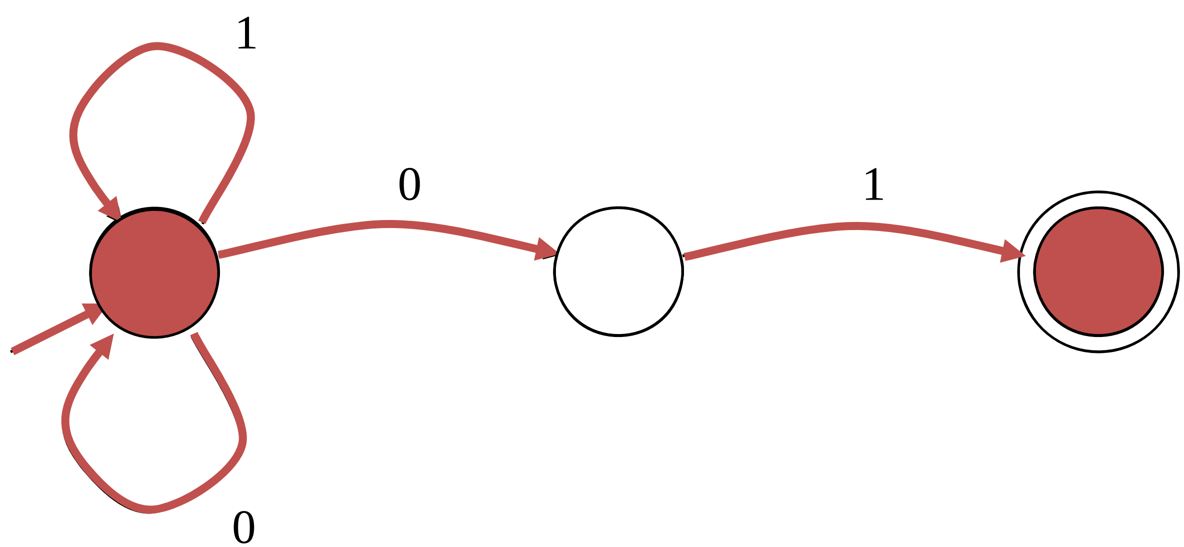
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

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 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

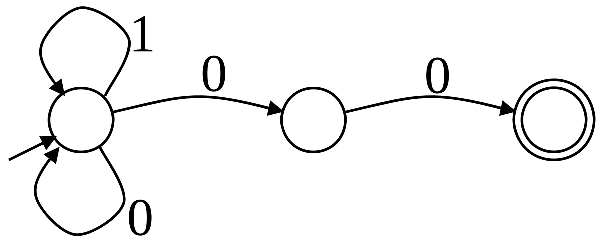
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

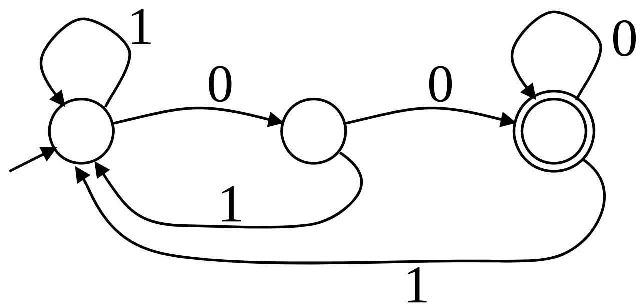
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



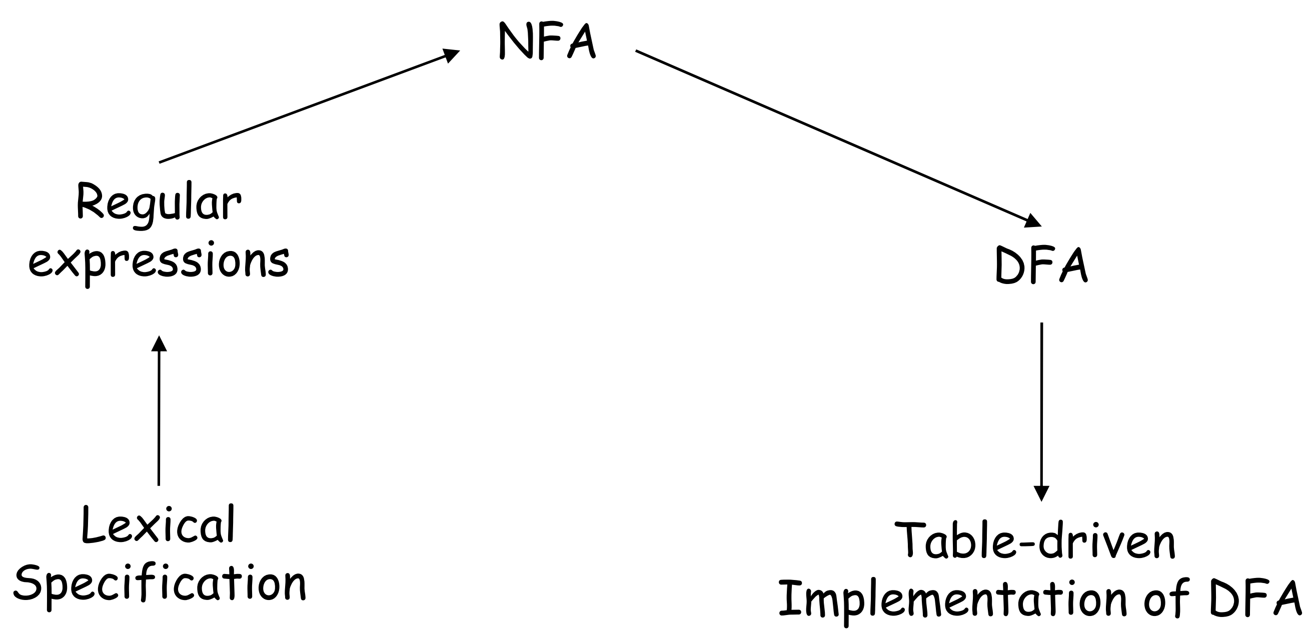
DFA



- DFA can be exponentially larger than NFA

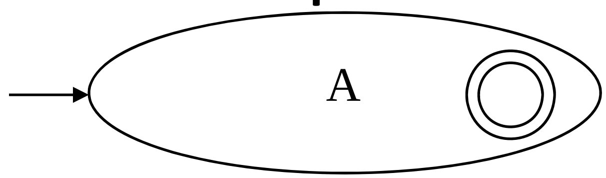
Regular Expressions to Finite Automata

- High-level sketch

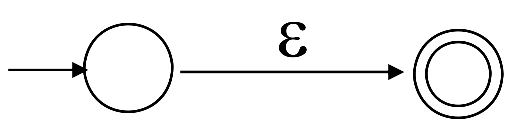


Regular Expressions to NFA (1)

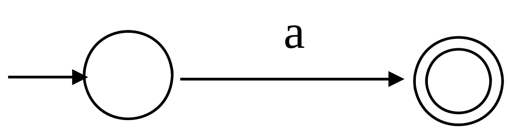
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A
- Decomposite rexp into sub-rexp.



- For ε

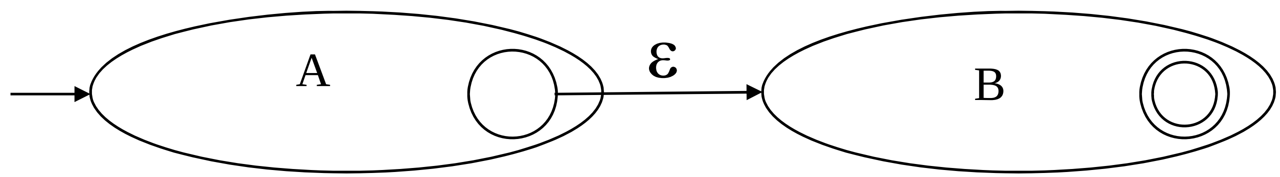


- For input a

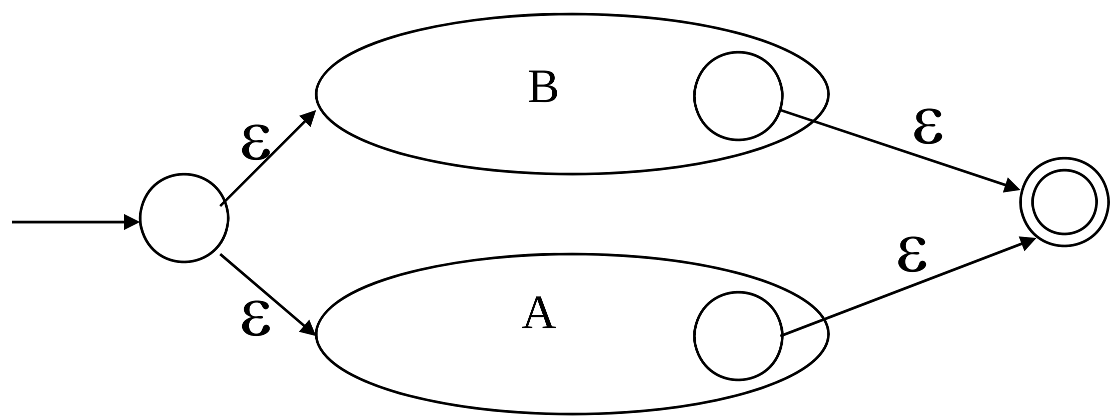


Regular Expressions to NFA (2)

- For AB

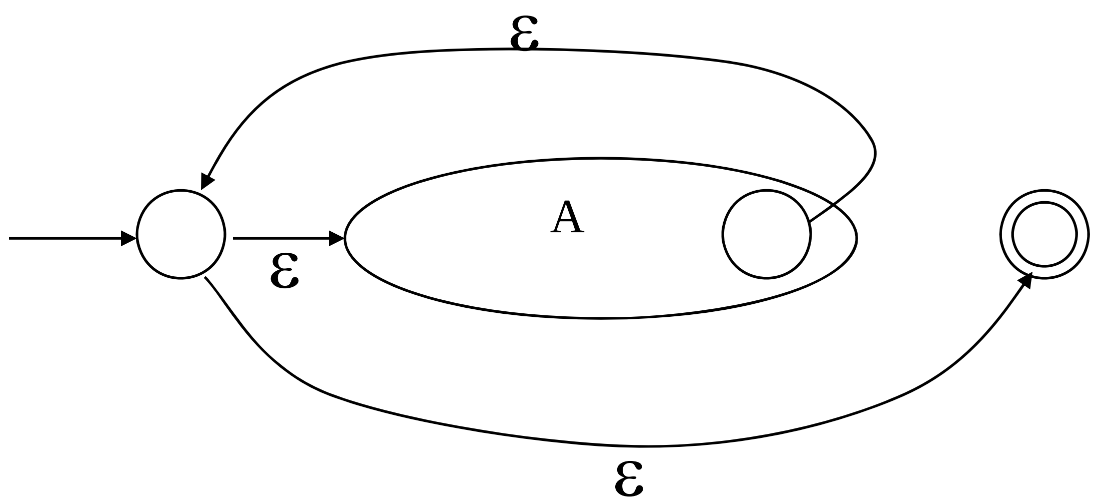


- For *A* | *B*



Regular Expressions to NFA (3)

- For A*



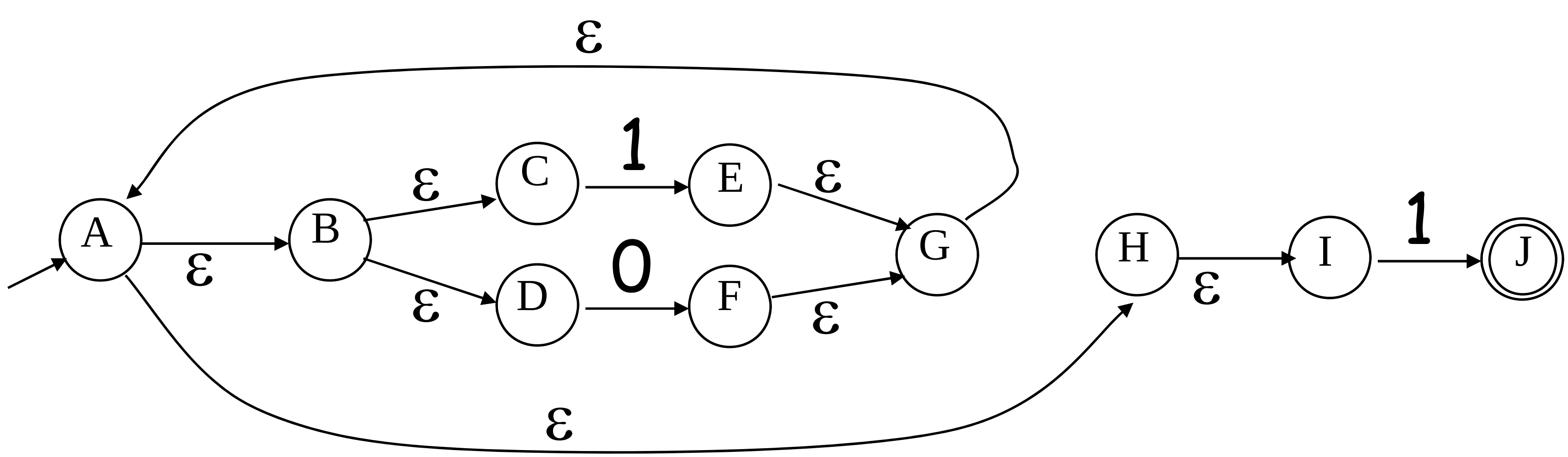
- For (A): same as A

Example of RegExp -> NFA conversion

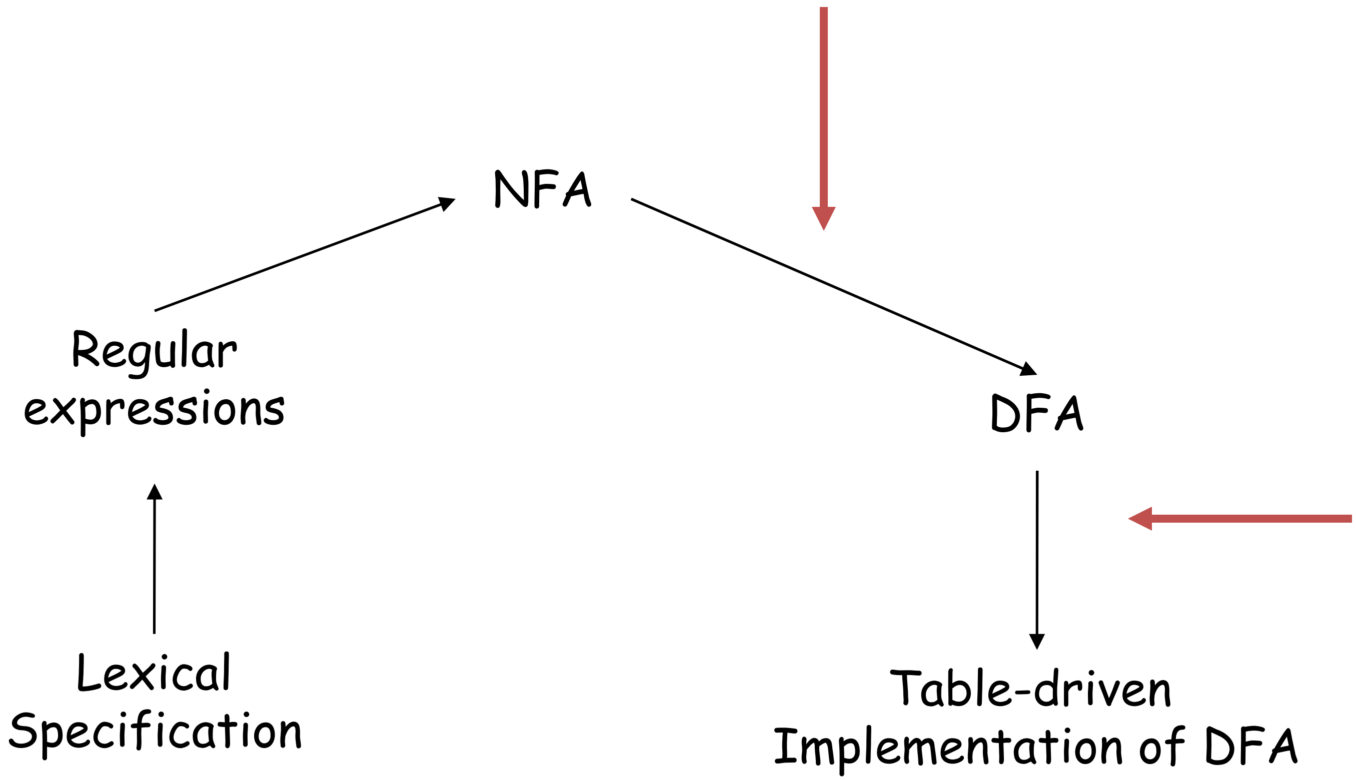
- Consider the regular expression

  (1 | 0)*1

- The NFA is

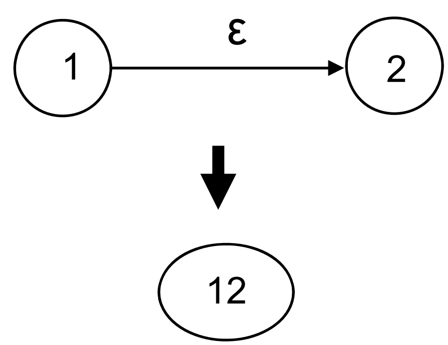


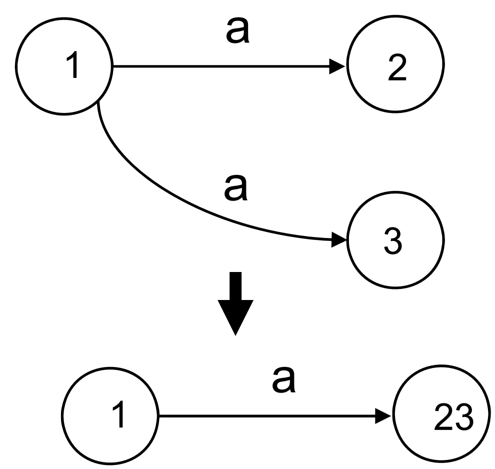
Next


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

Basic ideas of remove non-determinism

- Two cases of non-determinism:
  - Epsilon transition
    - Remove the edge by merging the two states
  - Exiting from one state there are multiple edges with same labels.
    - Merge the states that can be reached from the same symbol;





NFA to DFA. The Trick

- Simulate the NFA
- Each state of DFA

  = a non-empty subset of states of the NFA
- Start state

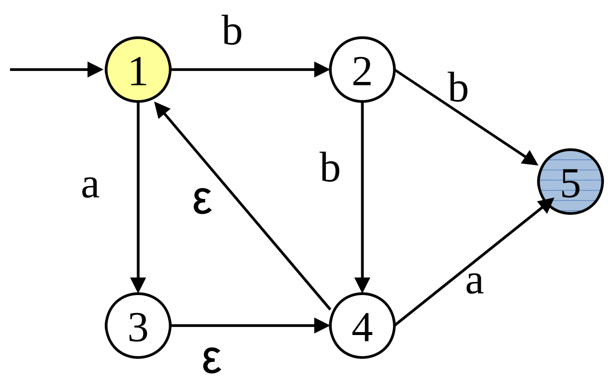
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

Formalize the ideas

- Two key functions
  - *ε-closure(T)* is set of states reachable by ε from s_i in T;
  - *Move(T,a)* is set of states reachable by a from s_i in T.
- The algorithm
  - Start state derived from s_0 of the NFA
  - Take its ε-closure
  - Work outward, trying each $\alpha \in \Sigma$ and taking its ε-closure
  - Each state in DFA corresponds to a *subset* of states of the NFA;
  - That is why it is called subset construction;
  - Iterative algorithm that halts when the states wrap back on themselves.

ε-closure

- Definition: ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.
- Example:



ε -closure({1,2,5}) = {1,2,5}
ε -closure({4}) = {1,4}
ε -closure({3}) = {1,3,4}
ε -closure({3,5}) = {1,3,4,5}

NFA to DFA. Algorithm(I)

- A transition table Dtran for D is constructed as follows:

initially, ε-closure(s_0) is the only state in *Dstates* and it's unmarked;

while there is an unmarked state in T in *Dstates* **do begin**

    mark T;

    for each input symbol a **do begin**

        U := ε-closure(move(T,a));

        if U is not in *Dstates* **then**

            add U as an unmarked state to *Dstates*;

            Dtran[T,a] := U

        end

end

NFA to DFA. Algorithm(II)

- Computation of ε-closure(T) :

push all states in T onto *stack*;

initialize ε-closure(T) to T;

while *stack* is not empty **do begin**

  pop t, the top element, off of *stack;*

  for each state u with an edge from t to u labelled ε **do begin**

    if u is not in ε-closure(T) **then**

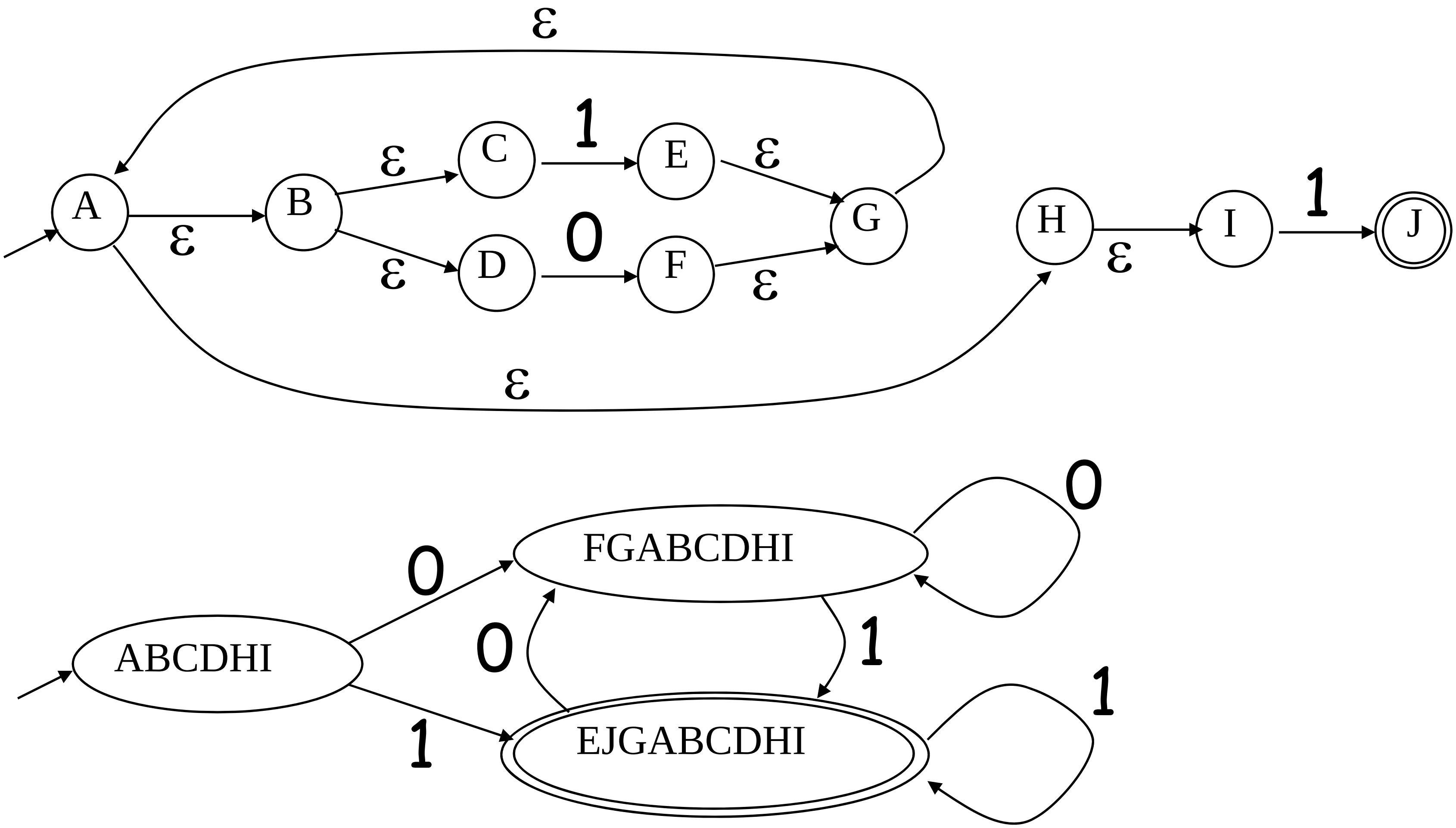
      add u to ε-closure(T);

      push u onto stack;

    end

end

NFA -> DFA Example



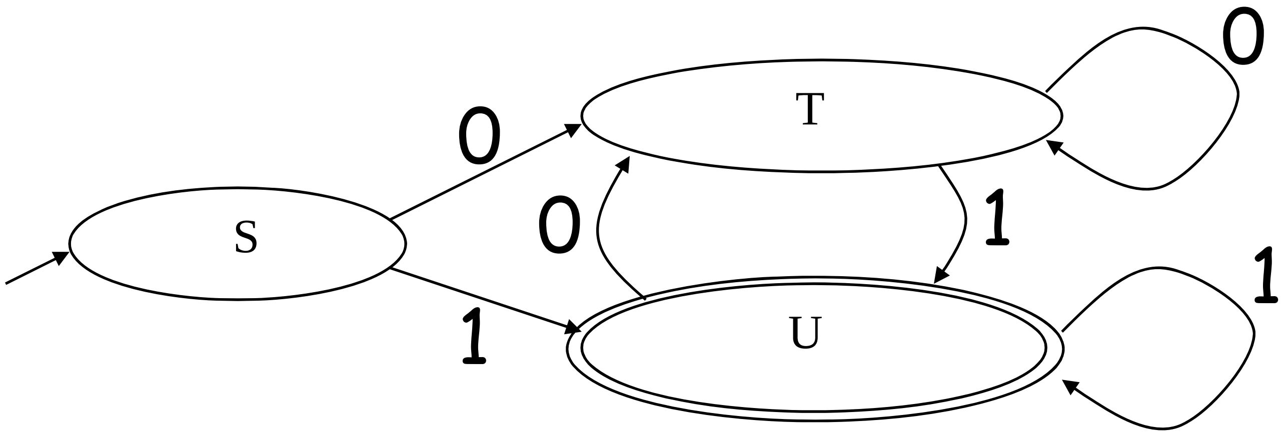
Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as lex, flex or jlex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

Table Implementation of a DFA

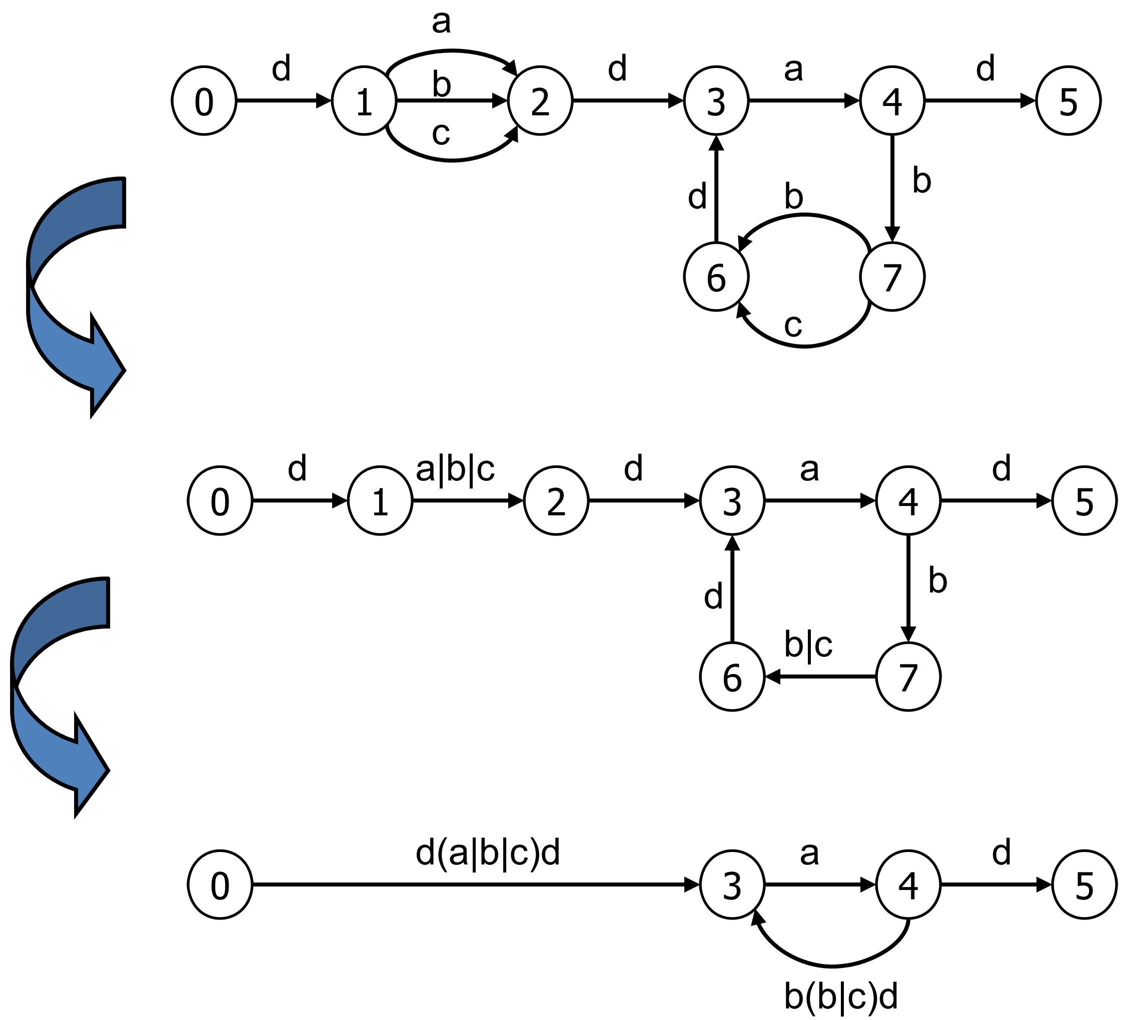


	0	1
S	T	U
T	T	U
U	T	U

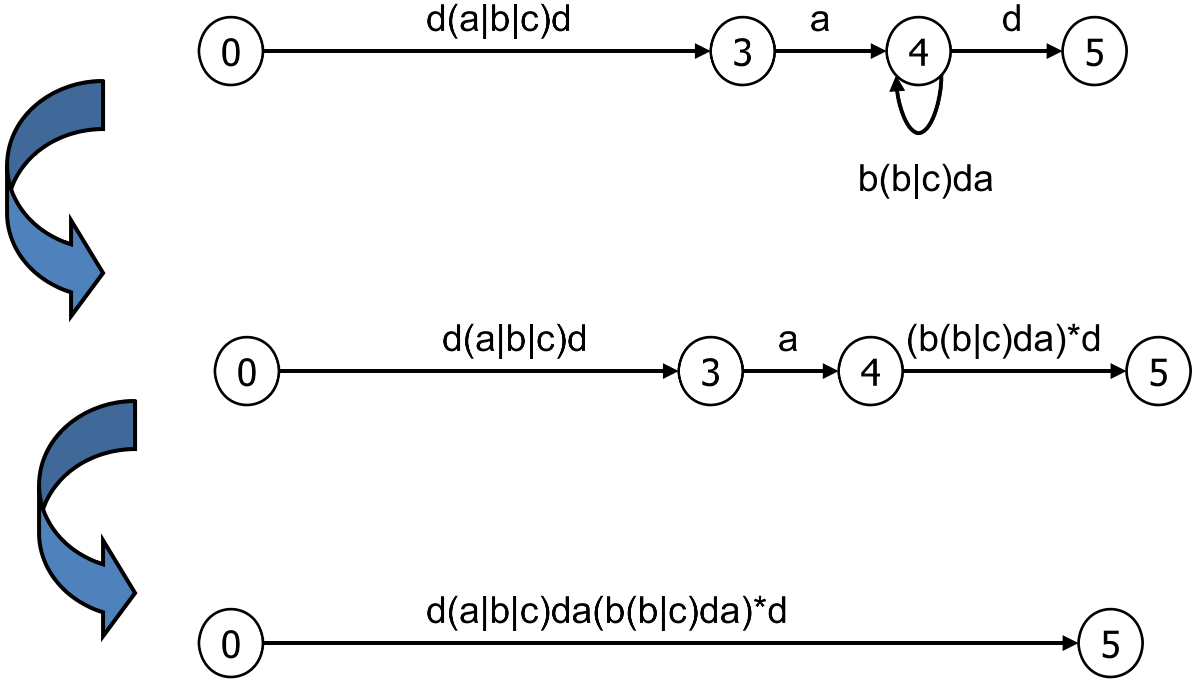
Converting DFAs to REs

1. Combine serial links by concatenation
2. Combine parallel links by alternation
3. Remove self-loops by Kleene closure
4. Select a node (other than initial or final) for removal. Replace it with a set of equivalent links whose path expressions correspond to the in and out links
5. Repeat steps 1-4 until the graph consists of a single link between the entry and exit nodes.

Example



Example (cont.)



Scanner generator: history

- LEX
  - Original for UNIX, it now exists for many operating systems;
  - LEX produces a scanner which is a C program;
  - LEX accepts regular expressions and allows actions (i.e., code to executed) to be associated with each regular expression.
- JLex
  - Lex that generates a scanner written in Java;
  - Itself is also implemented in Java.

```
%{
    #include <stdio.h>
    int num_lines = 0, num_chars = 0;
%}
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
    yylex();
    printf( "# of lines = %d, # of chars = %d \n",num_lines,
    num_chars );
}
```

```
%{
#include <stdio.h>
%}
%%
[0-9]+  {
            /* yytext is a string containing the matched text. */
            printf("Saw an integer: %s\n", yytext);
        }
.       {   /* Ignore all other characters. */   }
%%
int main(void)
{
   yylex();
   return 0;
}
```