Regular Languages & Finite Automata

- Basic formal language theory result:

Regular expressions and finite automata both define the class of regular languages.

Aspect	**Regular Expression**	**Finite Automaton**
Used For	Specification (what pattern to match)	Implementation (how to match it)
Form	Symbolic pattern	State-transition diagram or table
Conversion	RE → NFA → DFA	DFA → RE (via state elimination)
Tool Use	Lexer specs (Flex, Lex)	Actual automaton used for tokenizing

Thus, we are going to use:

Regular expressions for specification

Finite automata for implementation

(automatic generation of lexical analyzers)

Finite Automata

- A finite automaton is a recognizer for the strings of a regular language

- A finite automaton consists of
  - A finite input alphabet **Σ**
  - A set of states **S**
  - A start state **n**
  - A set of accepting states **F ⊆ S**
  - A set of transitions **state** $_i \xrightarrow{input}$ **state** $_j$

Finite Automata

- Transition

$$s_1 \to^a s_2$$

- Is read

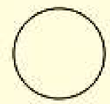
  In state s_1 on input "a" go to state s_2

- If end of input
  - If in accepting state ⇒ accept, Otherwise ⇒ reject

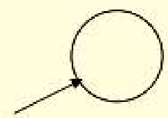
- If No transition possible => Reject

Finite Automata State Graphs

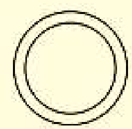
- A state



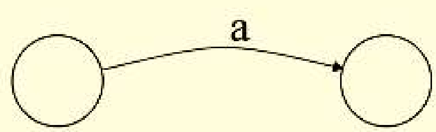
- The start state



- An accepting state



- A transition



Transition Diagrams

- Regular expression are declarative specifications
- Transition diagram is an implementation

  - An input alphabet belonging to Σ
  - A set of states S
  - A set of transitions $\text{state}_i \rightarrow^{input} \text{state}_j$
  - A set of final states F
  - A start state n

  Transition $s1 \rightarrow^{a} s2$ is read:

  - If end of input is reached in a final state, then accept
  - Otherwise, reject

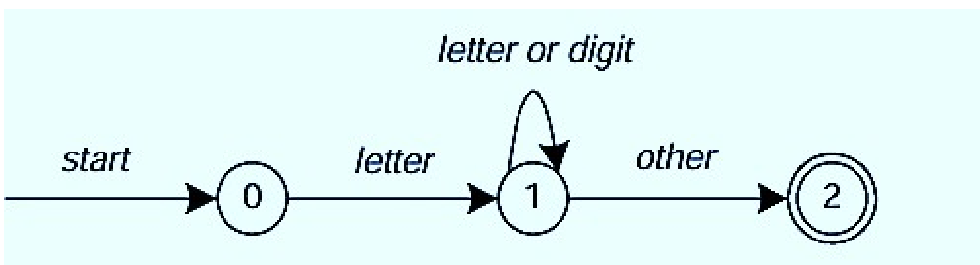
How to recognize tokens?

- Identifier: strings of letters or digits, starting with a letter

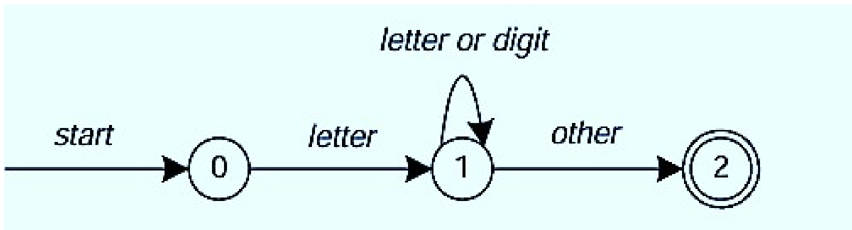
letter(letter|digit)*

- repetition, expressed by the "*" operator
- alternation, expressed by the "|" operator
- concatenation

- Any regular expression may be expressed as a finite state automaton (FSA).
- There is one start state and one or more final or accepting states.



Contd.,



→→→→→

```
start:   goto state0

state0:  read c
         if c = letter goto state1
         goto state0

state1:  read c
         if c = letter goto state1
         if c = digit goto state1
         goto state2

state2:  accept string
```

Finite State Automaton

3-state machine

How to recognize tokens

- Consider

relop → < | <= | = | <> | >= | >

id → letter(letter|digit)*

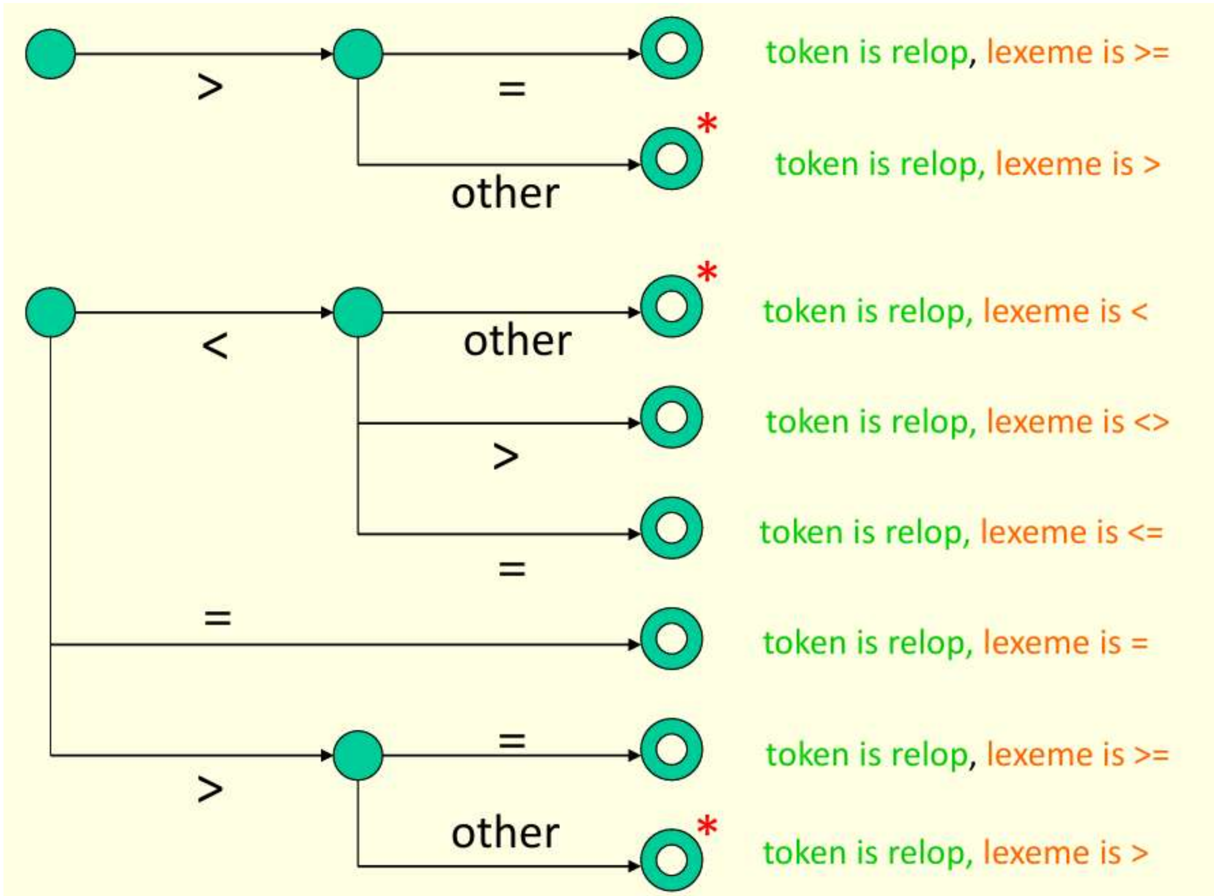
num → $digit^+$ (‘.’ $digit^+$)? (E(‘+’|’-’)? $digit^+$)?

delim → blank | tab | newline

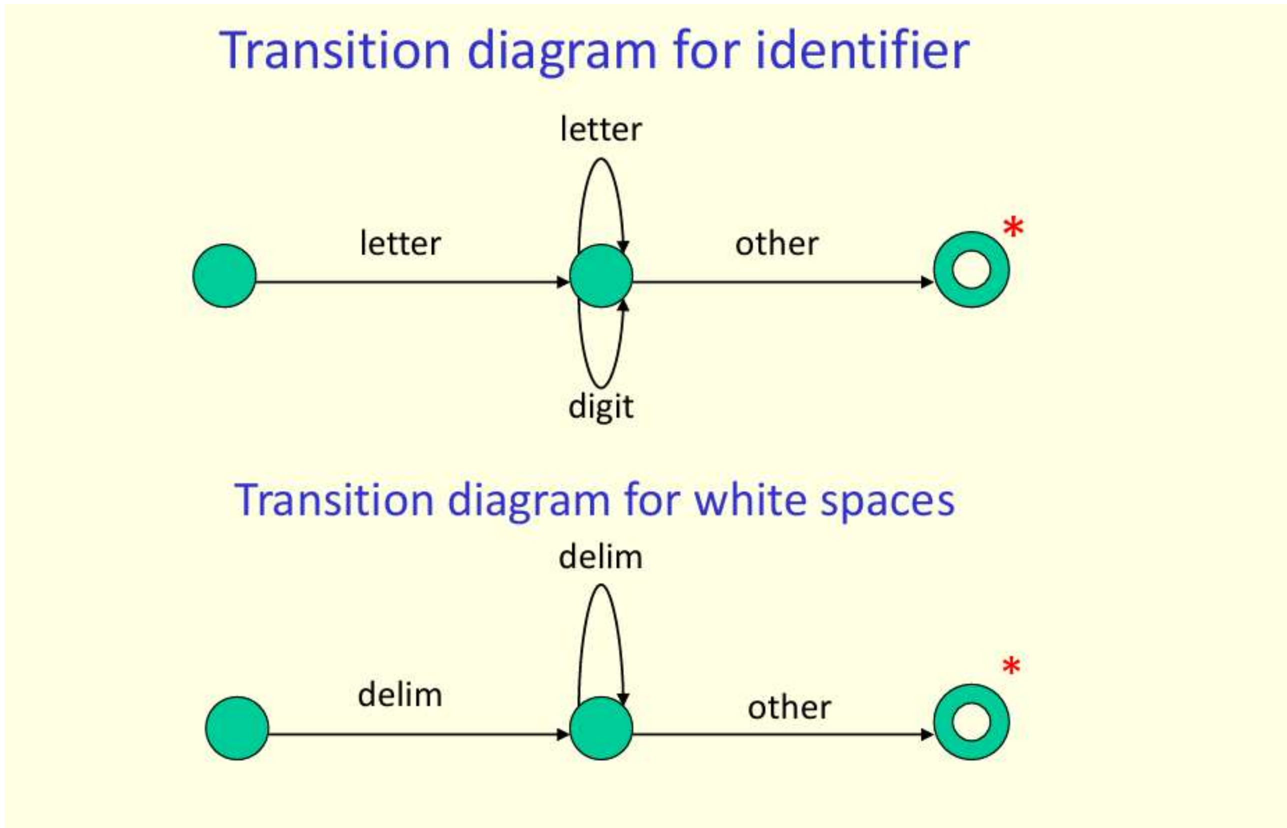
ws → $delim^+$

Construct an analyzer that will return <token, attribute> pairs

Transition diagram for relational operators

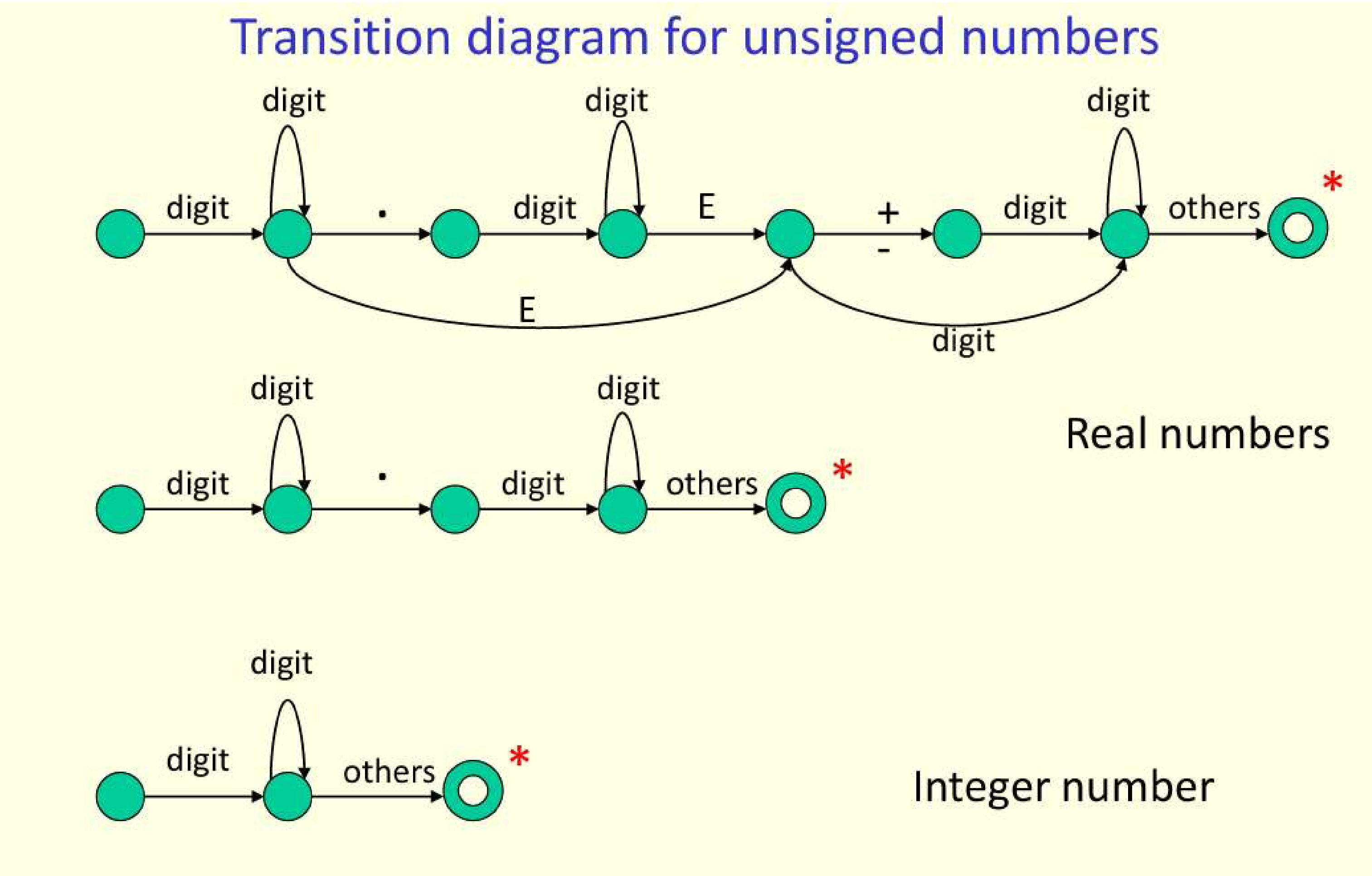


Contd.,



delim →→→→ blank | tab | newline

Contd.,



Recognizes real numbers with exponential notation.

Ex: **12.34E+56**

←←←←←←←←

Does **not** include exponential form.

Ex: **12.34**

←←←←←

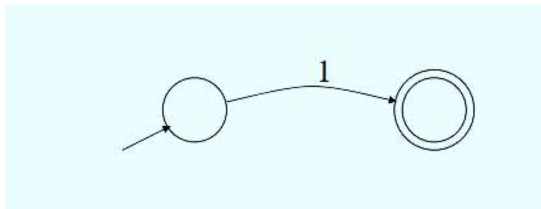
Simple DFA recognizing a sequence of digits followed by a non-digit.

Ex: **1234**

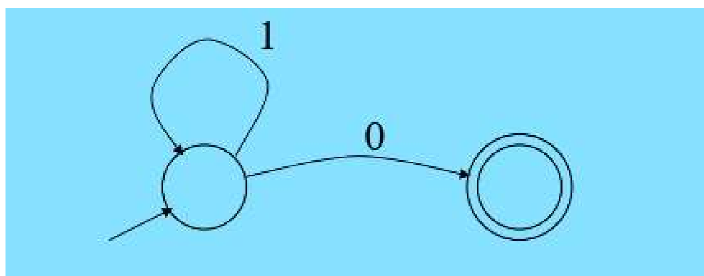
←←←←←←

A Simple Example

- A finite automaton that accepts only “1”.



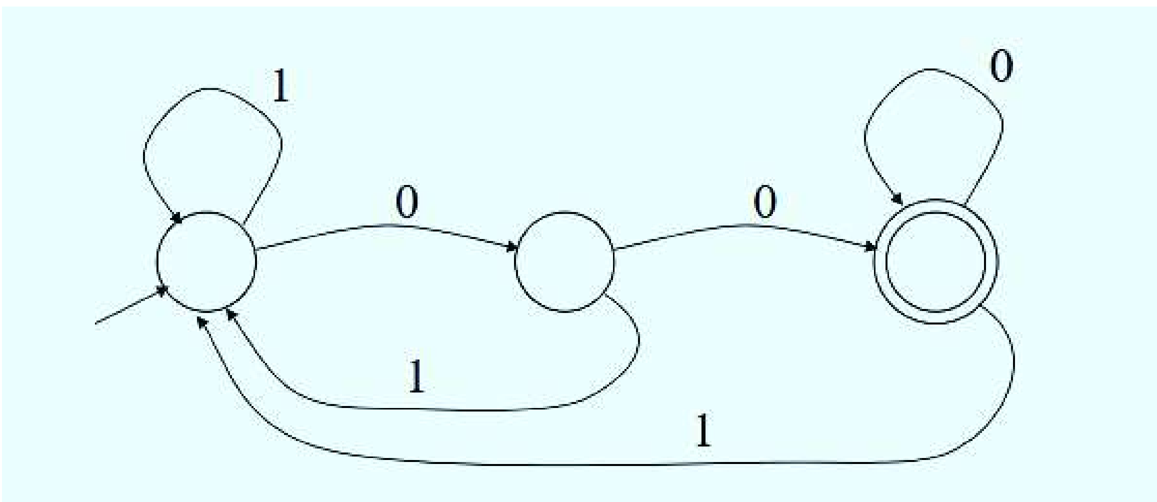
- Another Simple Example
  - A finite automaton accepting any number of 1’s followed by a single 0
  - Alphabet: {0,1}



Check that “1110” is accepted but “110...” is not

And Another Example

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

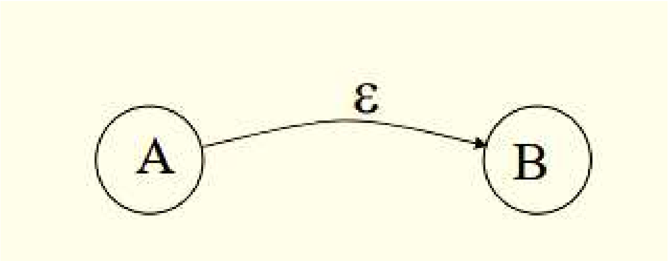


{0, 1}*, it means it **accepts every possible binary string**, including the empty string.

ε, 0, 1, 00, 01, 10, 11, 000, 001, ..., 101010, etc.

Epsilon Moves

- Another kind of transition: ε-moves



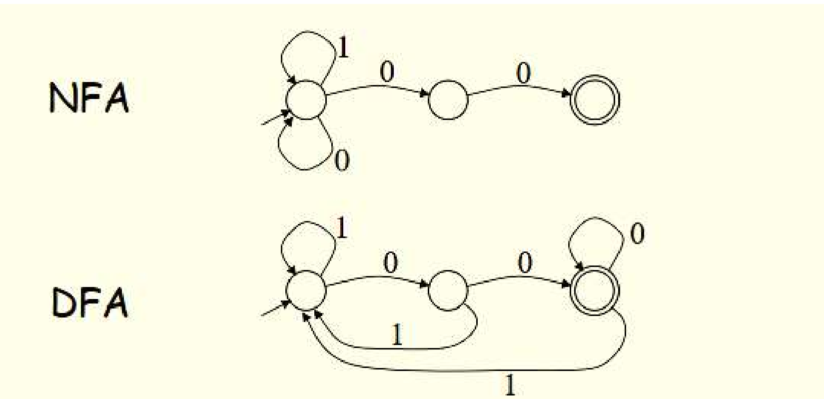
- Machine can move from state A to state B without reading input

Deterministic and Non-Deterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- Finite automata have finite memory
  - Enough to only encode the current 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
- For a given language the NFA can be simpler than the DFA



Accepts strings like: "0 0", "1 0 0", "0 1 0", etc.

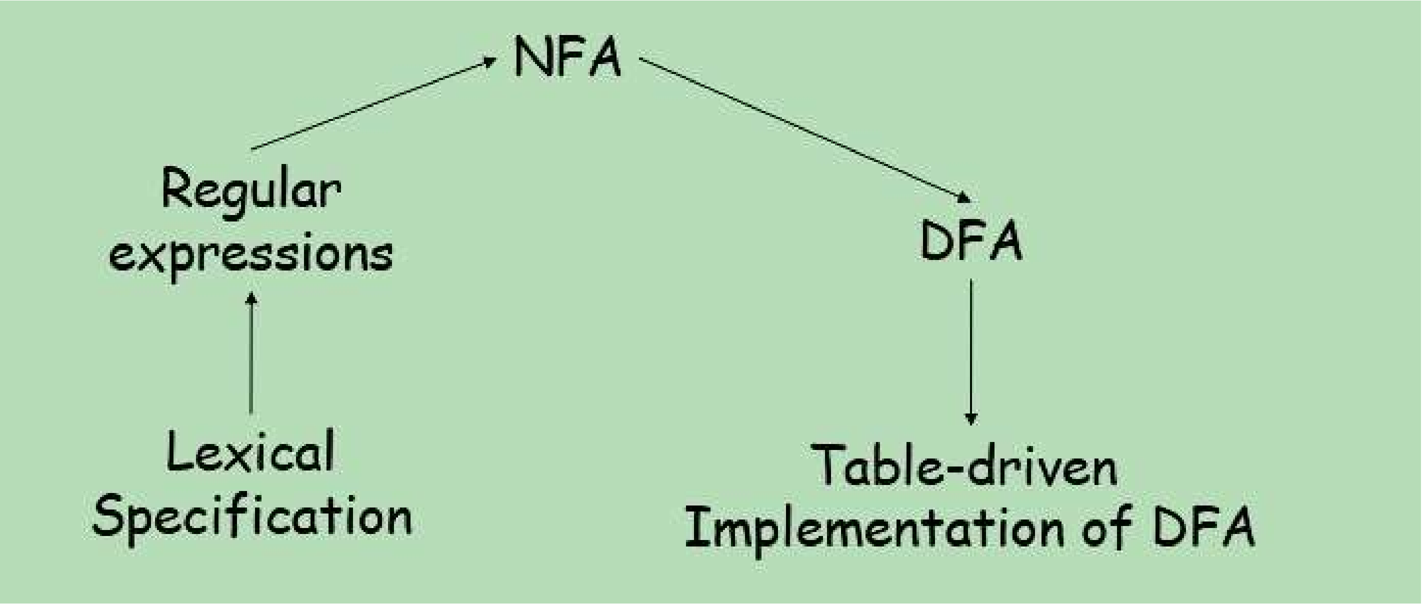
Each input at each state, Handles all possibilities deterministically.

Same language accepted, but every state has exactly one transition for each symbol (0 or 1).

- DFA can be exponentially larger than NFA

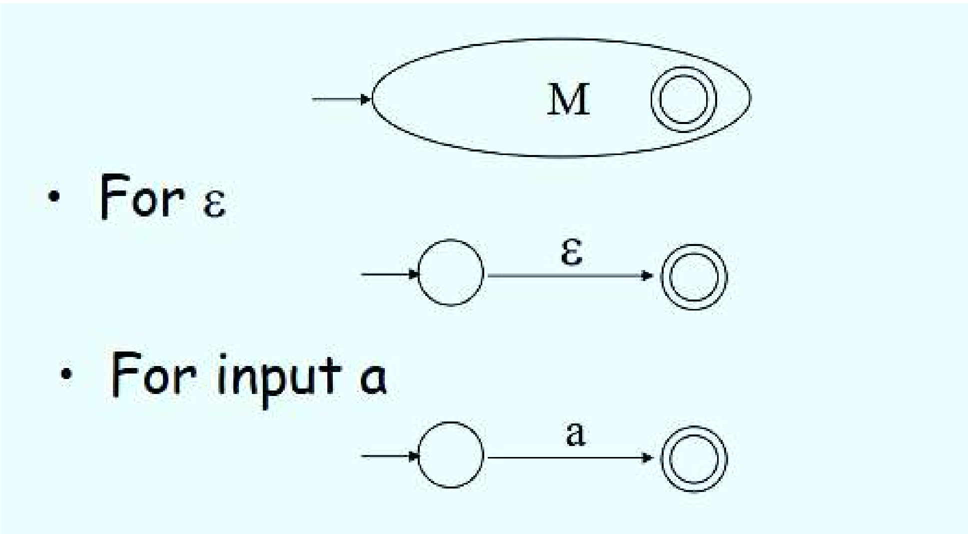
Regular Expressions to Finite Automata

- High-level sketch



Regular Expressions to NFA (1)

- For each kind of reg. expr, define an NFA: **Basic NFA fragments used in Thompson's construction** of regular expressions



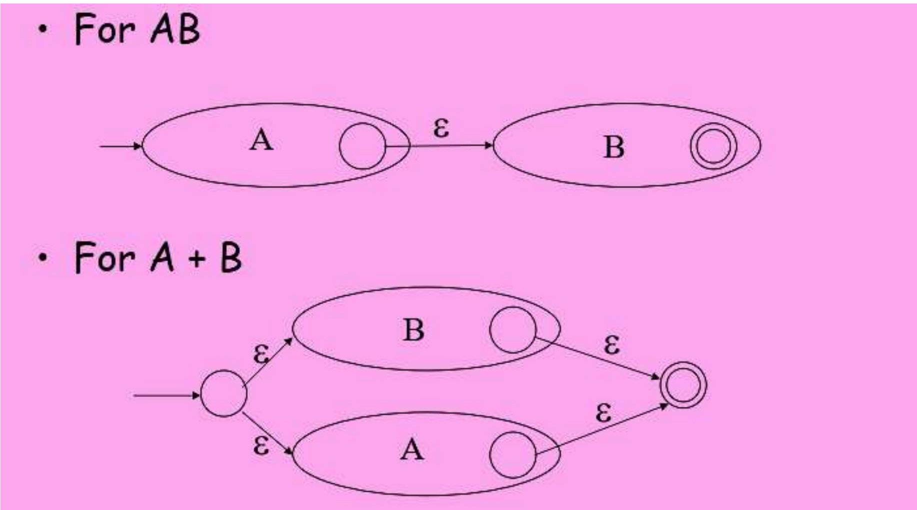
- M represents a **general NFA fragment**

ε-NFA Fragment (For ε)

- Two states: a start state and a final state with an ε-transition (epsilon move) between them.
- This NFA accepts the **empty string (ε)** — i.e., without consuming any input character.

Basic Symbol NFA (For input a): This NFA accepts **only the string "a"**.

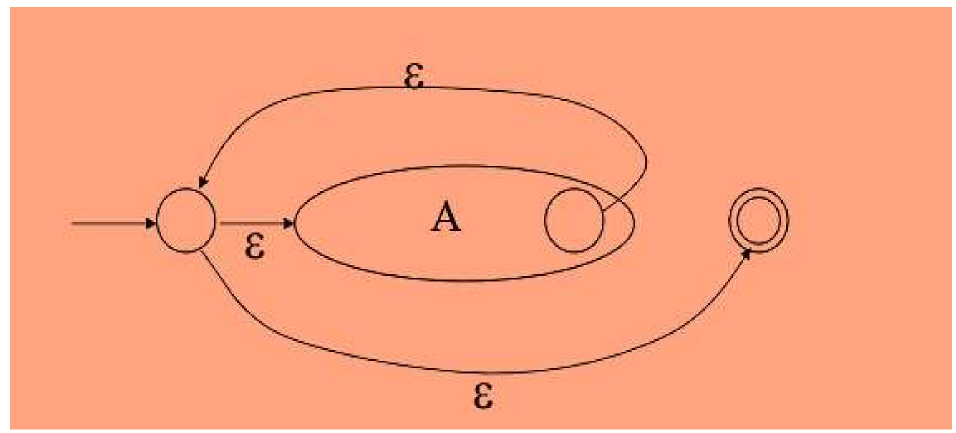
Regular Expressions to NFA (2)



In **Thompson's construction**, each regular expression is broken into:

- Basic parts like a, b, ε,
- And combined using operations: **Concatenation** (ab), **Union** (a|b), **Kleene star** (a*)

Regex	Meaning	NFA Mechanism
AB	Concatenation	A →ε→ B
A + B	Union	ε→A and ε→B; A→ε→Final, B→ε→Final
A*	Zero/more of A	Loop: Start→ε→A→ε→Start; ε→Final for empty

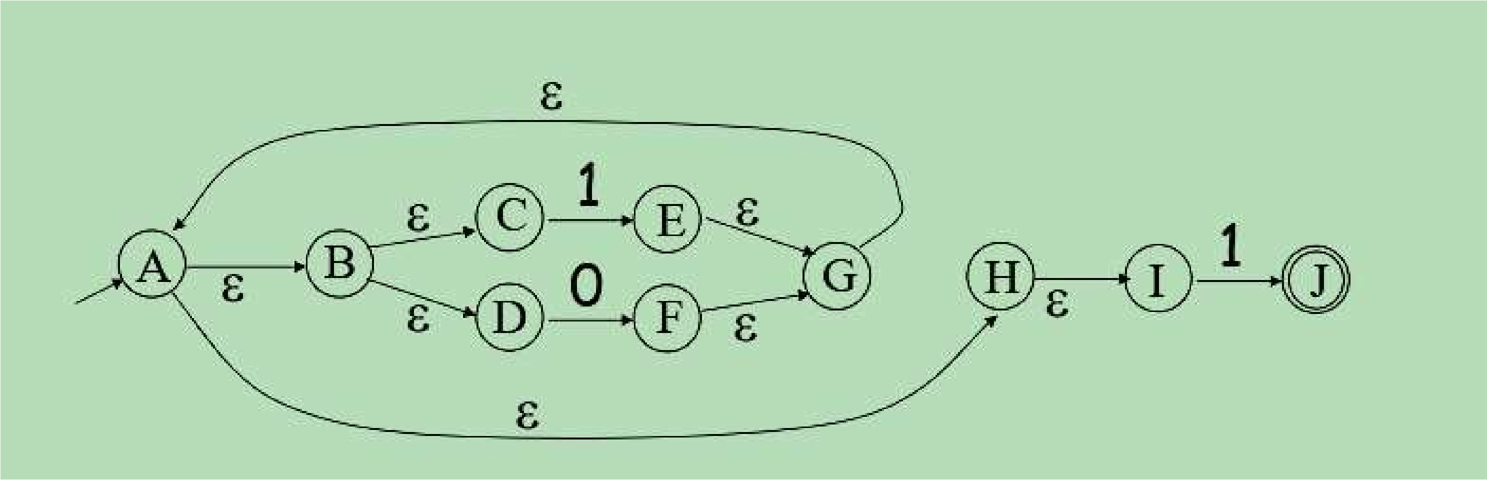


For A*
→→→→→

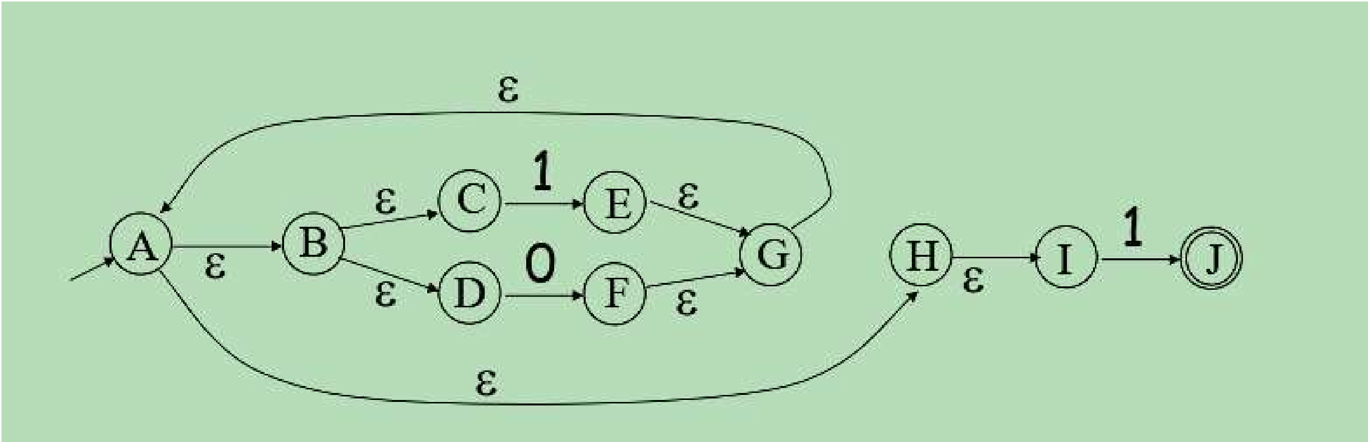
Example of Regular Expression → NFA conversion

- Consider the regular expression: (1+0)*1

The NFA is



Thompson's Construction → ε-NFA



- **A** is the initial state
- Loop for (1 + 0)*:
  - Through ε-transitions, it branches to:
    - B → ε → C → 1 → E → ε → G
    - B → ε → D → 0 → F → ε → G
  - Then G → ε → A to allow repetition.
- Final part A → ε → H → ε → I → 1 → J (final)
- So it accepts any combination of 0s and 1s followed by a 1.
- This is the ε-NFA formed using **Thompson's Construction**.

Subset Construction → DFA: 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 →a S' to DFA iff
  - S' is the set of NFA states reachable from any state in S after seeing the input a
    - considering ε-moves as well

Example

Given ε-NFA → States

- q_0 (start state)
- q_1
- q_2

Transitions:

- $q_0 \xrightarrow{\epsilon} q_1$
- $q_1 \xrightarrow{\epsilon} q_2$
- $q_0 \xrightarrow{a} q_0$
- $q_1 \xrightarrow{b} q_1$

ε-Closures: The **ε-closure** of a state is the set of states reachable from it using **only ε-transitions**, including the state itself.

1. **ε-closure(q_0)** = $\{q_0, q_1, q_2\}$

   ($q_0 \rightarrow q_1 \rightarrow q_2$ through ε-transitions)

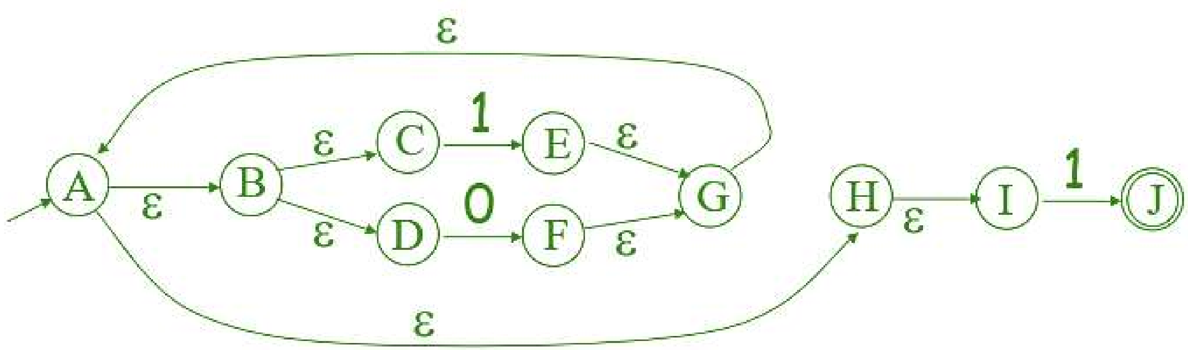
2. **ε-closure(q_1)** = $\{q_1, q_2\}$

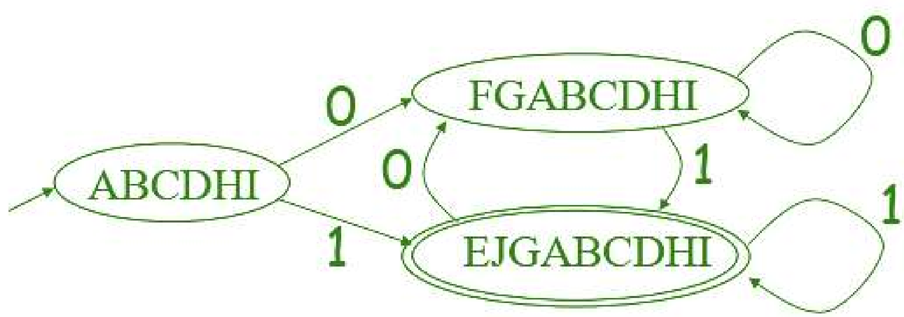
   ($q_1 \rightarrow q_2$ through ε)

3. **ε-closure(q_2)** = $\{q_2\}$

   (no ε-transitions from q_2)

NFA to DFA Example





Step 1: ε-closure(A)
From A:

- A → ε → B, H
- B → ε → C, D
- C → 1 → E
- D → 0 → F
- H → ε → I
- (E, F → ε → G), G → ε → A
- I → 1 → J

ε-closure(A) = {A, B, C, D, H, I}

- **On input 0 from S_0 = {A, B, C, D, H, I}:**
- **Only D has a 0-transition: D → 0 → F**
- **Then:**
  - **F → ε → G**
  - **G → ε → A**
  - **From A → ε → {B, C, D, H, I} → and it loops**

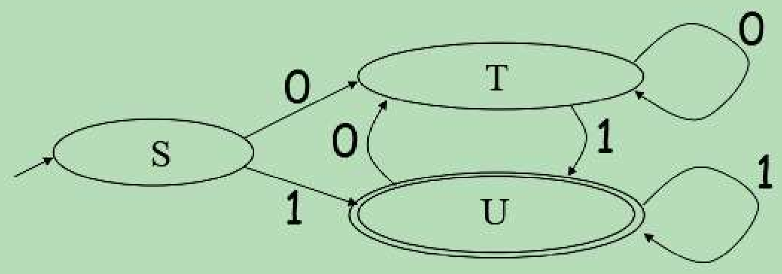
ε-closure(F) = {F, G, A, B, C, D, H, I}

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition **state** $_i$ $\xrightarrow{input(a)}$ **state** $_k$ define T[i,a] = k

- DFA "execution"
  - If in **state** $_i$ and input a, read T[i,a] = k and skip to state **state** $_k$ Sk
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

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

Theory vs Practice

- Two differences:
  - DFAs recognize lexemes.
    - A lexer must return a type of acceptance (token type) rather than simply an accept/reject indication.
  - DFAs consume the complete string and accept or reject it.
    - A lexer must find the end of the lexeme in the input stream and then find the next one, etc.

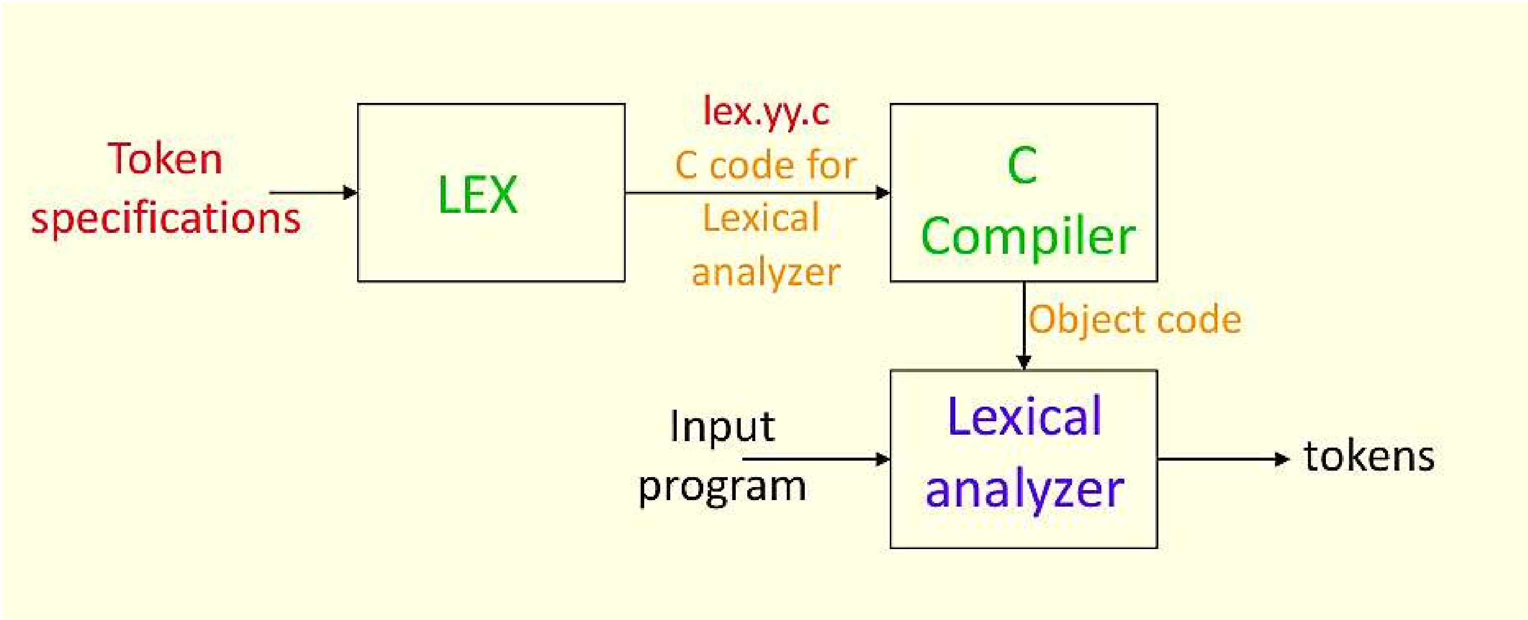
Lexical analyser generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression

    (generates kind of token and other keeping information)

- Output of the generator
  - Program that reads input character stream and breaks that into tokens
  - Reports lexical errors (unexpected characters), if any

LEX: A lexical analyzer generator



How does LEX work?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non deterministic finite automaton (NFA)
- Convert the NFA into an equivalent DFA
- Minimize the DFA to reduce number of states
- Emit code driven by the DFA tables