COMPILERS

LECTURE 3
LEXICAL ANALYSIS – FINITE AUTOMAT

Dr. Ahmed Hesham Mostafa

Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called transition diagrams."
- we perform the conversion from regular-expression patterns to transition diagrams by hand, we shall Finite Automata to construct these diagrams from collections of regular expressions.
- Transition diagrams have a collection of nodes or circles, called states.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Example :Recognition of tokens

The next step is to formalize the patterns:

- *digit* -> [0-9]
- *Digits* -> digit+
- *number* -> digit(.digits)? (E[+-]? Digit)?
- *letter* -> [A-Za-z_]
- *id* -> letter (letter|digit)*
- *If* -> if
- *Then* -> then
- *Else* -> else
- *Relop* -> < | > | <= | >= | = | <>

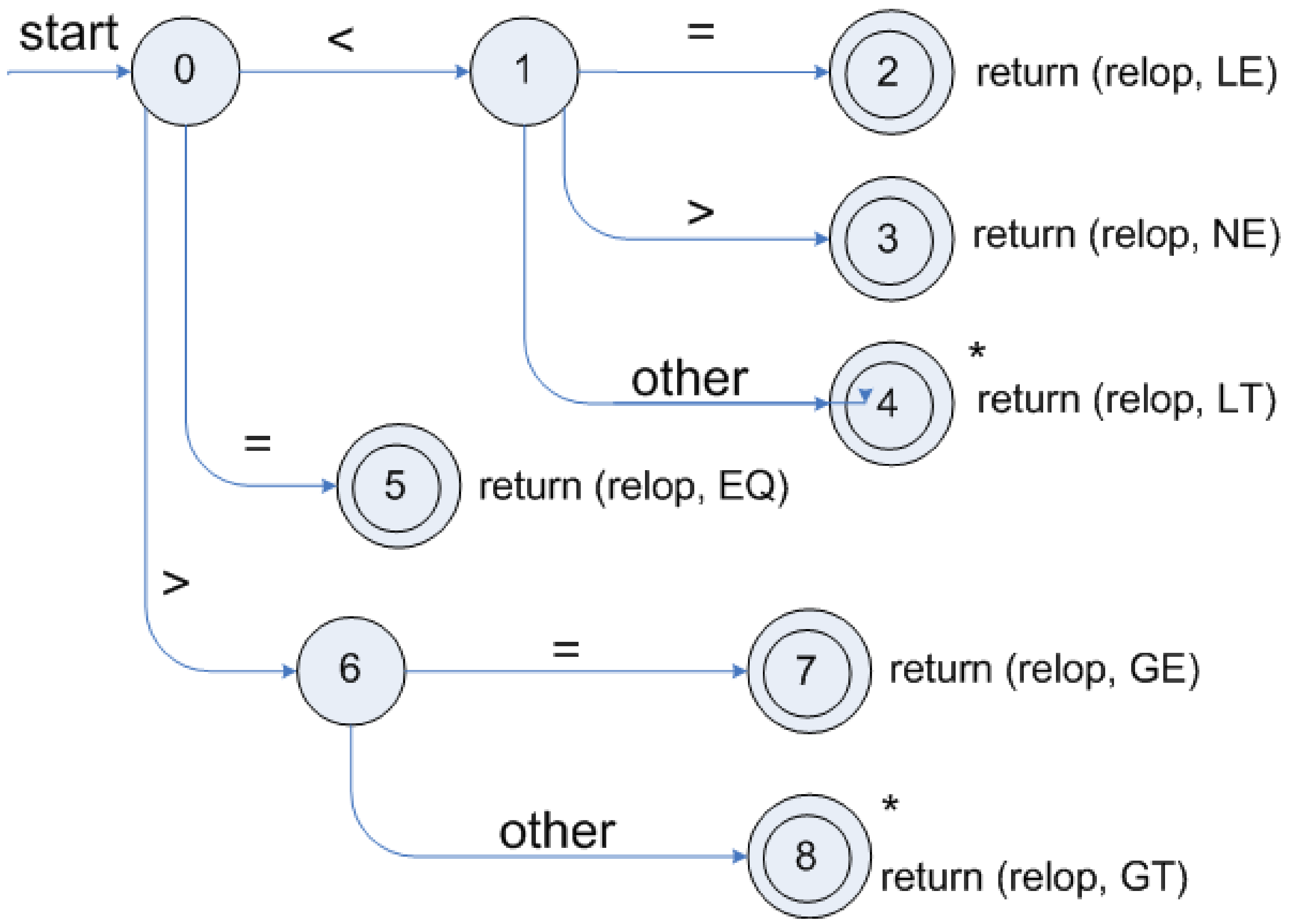
We also need to handle whitespaces:

- *ws* -> (blank | tab | newline)+

Lexemes	Token Name	Attribute Value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Transition diagrams

- Transition diagram for relop Fig. 3.13



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {            /* repeat character processing until a
                                   return or failure occurs          */
    switch(state) {
            case 0: c= nextchar();
                        if (c == ‘<‘) state = 1;
                        else if (c == ‘=‘) state = 5;
                        else if (c == ‘>’) state = 6;
                        else fail();/* lexeme is not a relop */
                        break;
            case 1: …
            …
            case 8: retract();
                       retToken.attribute = GT;
                       return(retToken);
    }
```

sketch of getRelop()

- **getRelop**(), a C++ function whose job is to simulate the transition diagram of Fig. 3.13 and return an object of type **TOKEN**, that is, a pair consisting of the token name (which must be **relop** in this case) and an attribute value (the code for one of the six comparison operators in this case).
- **getRelop()**first creates a new object **retToken** and initializes its first component to **RELOP**, the symbolic code for token **relop**.
- in case 0, the case where the current state is 0. A function **nextChar**() obtains the next character from the input and assigns it to local variable **c**.

sketch of getRelop()

- We then check **c** for the three characters we expect to find, making the state transition dictated by the transition diagram of Fig. 3.13 in each case.
- For example, if the next input character is =, we go to state 5. If the next input character is not one that can begin a comparison operator, then a function **fail()**is called.
- What **fail()**does depends on the global error-recovery strategy of the lexical analyzer.
- It should reset the forward pointer to **lexemeBegin**, in order to allow another transition diagram to be applied to the true beginning of the unprocessed input.

sketch of getRelop()

- It might then change the value of state to be the start state for another transition diagram, which will search for another token.
- Alternatively, if there is no other transition diagram that remains unused, **fail()** could initiate an error-correction phase that will try to repair the input and find a lexeme.

sketch of getRelop()

- Because state 8 bears a *, we must retract the input pointer one position (i.e., put c back on the input stream).
- That task is accomplished by the function **retract().** Since state 8 represents the recognition of lexeme >,
- we set the second component of the returned object, which we suppose is named attribute, to GT, the code for this operator.

Finite Automata

- Regular expressions = grammer
- 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

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

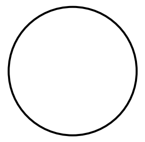
- Is read

In state s_1 on input “a” go to state s_2

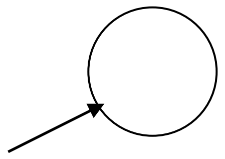
- If the end of input
  - If in accepting state => accept, otherwise => reject
- If no transition possible => reject

Finite Automata State Graphs

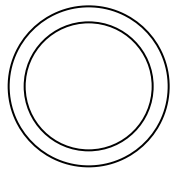
- A state



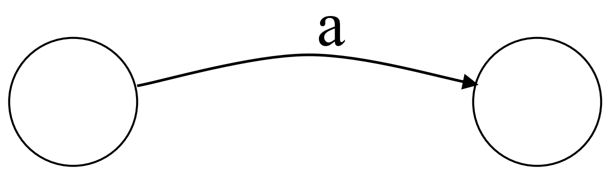
- The start state



- An accepting state

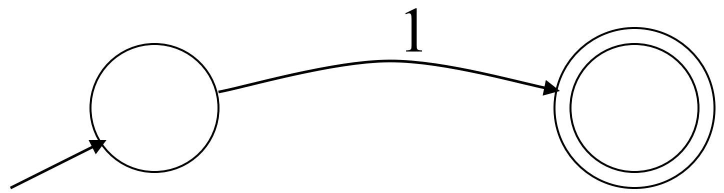


- A transition



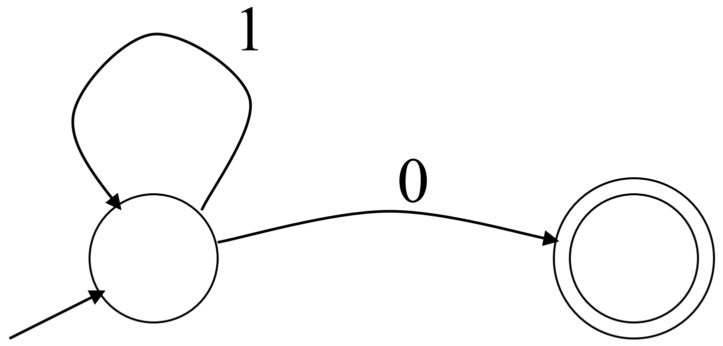
Example

- 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



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
- Examples of “110…” are “1101”, “1111000”,…



Deterministic and Nondeterministic Automata

Deterministic Finite Automata (DFA)

- One transition per input per state
- No ε-moves

Nondeterministic Finite Automata (NFA)

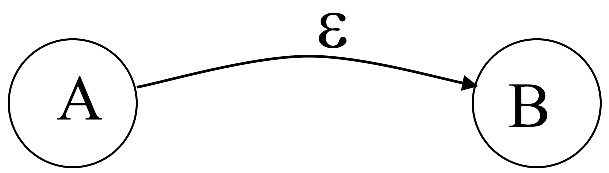
- Can have multiple transitions for one input in each state
- Can have ε-moves

Finite automata have *finite* memory

- Need only to encode the current state

Epsilon Moves(Transition)

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input
- "ε-transitions" do not process any characters, they just allow to "jump" between states

Execution of Finite Automata

A DFA can take only one path through the state graph

- Completely determined by input

NFAs can choose

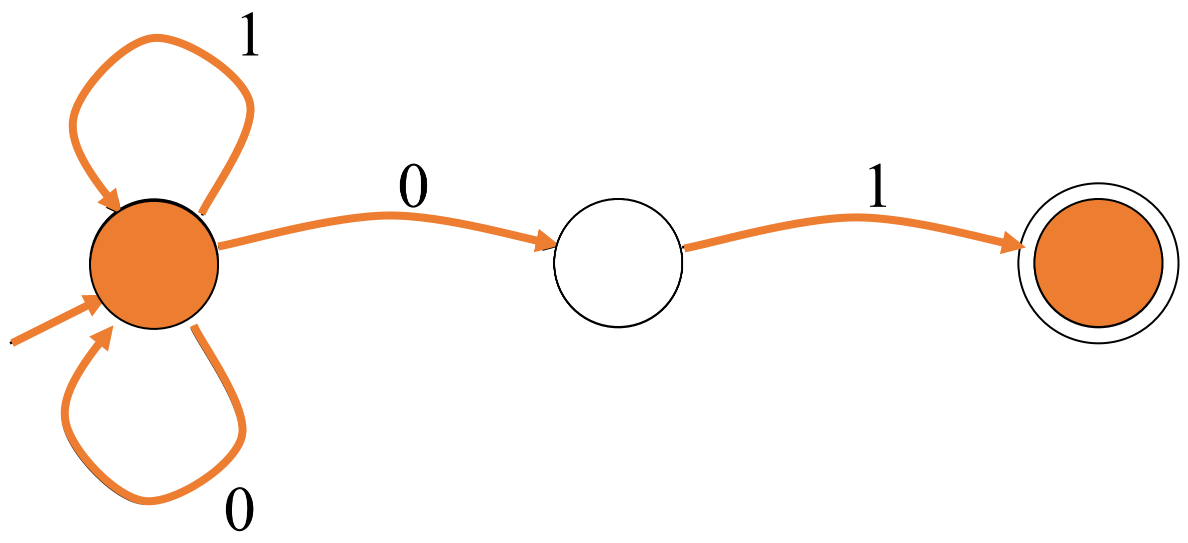
- Whether to make ε-moves
- Which of multiple transitions for a single input to take

NFA vs. DFA

- A NFA (nondeterministic finite automata) is able to be in several states at once.
    - In a DFA, we can only take a transition to a single deterministic state
    - In a NFA we can accept multiple destination states for the same input.
    - You can think of this as the NFA “guesses” something about its input and will always follow the proper path if that can lead to an accepting state.
    - Another way to think of the NFA is that it travels all possible paths, and so it remains in many states at once. As long as at least one of the paths results in an accepting state, the NFA accepts the input.

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

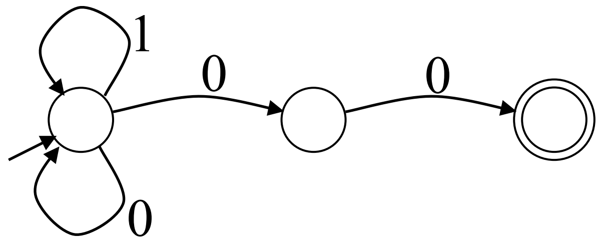
- NFAs and DFAs recognize the same set of languages (regular languages)

- NFA are easier to design
- NFA with ε-closure call **ε-NFA**
- DFAs are easier to implement
  - There are no choices to consider

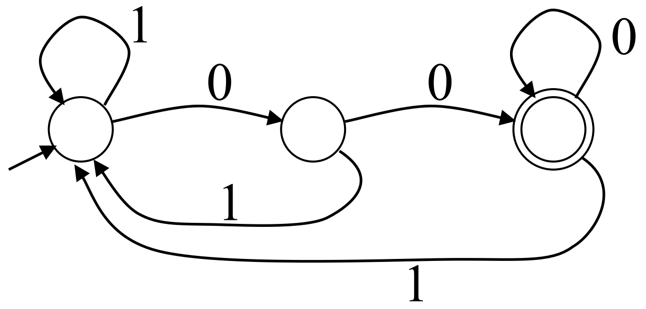
NFA vs. DFA

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

NFA

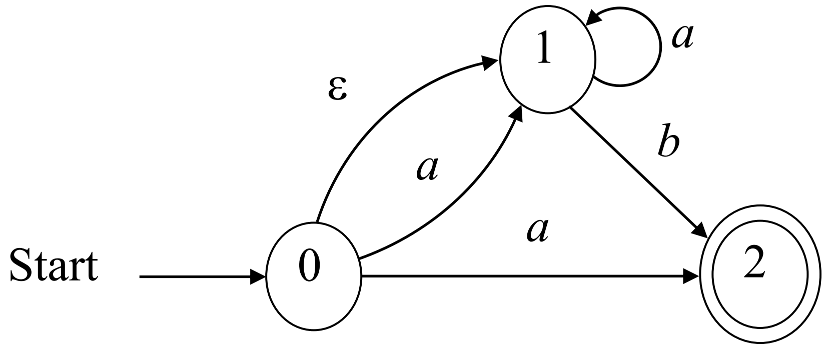


DFA



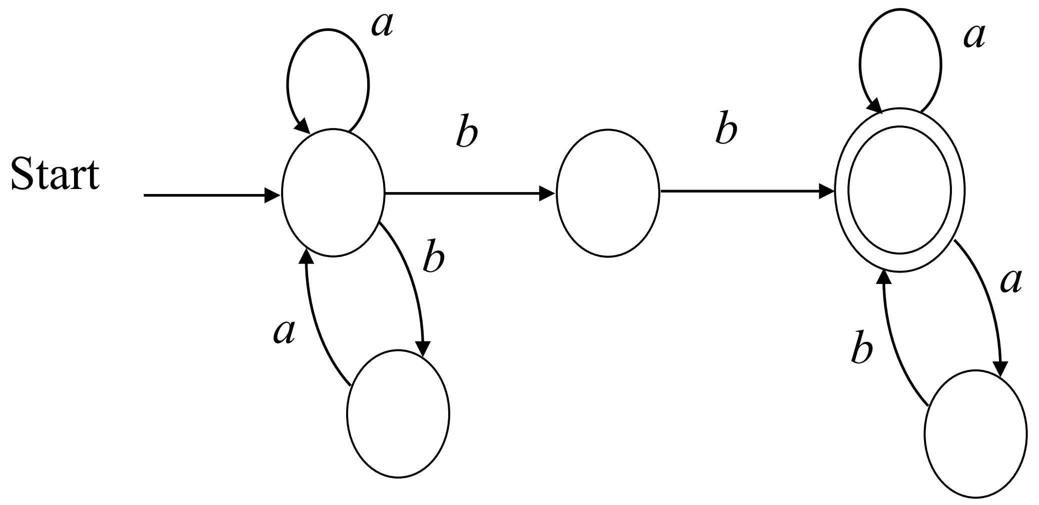
NFA Example

- The following NFA recognizes the language of regular expression
- (a | aa*b | a*b)



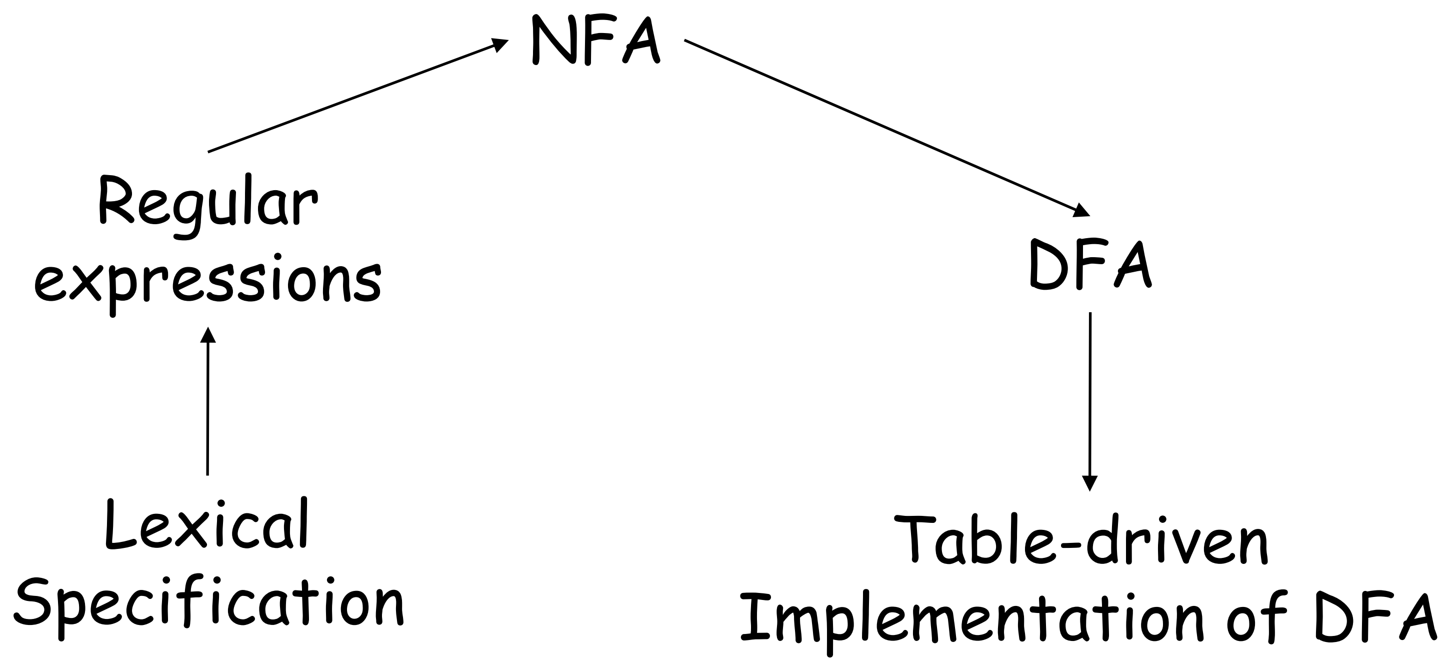
NFA Example

- The following NFA recognizes the language of regular expression
- (a | ba)*bb(a | ab)*



Regular Expressions to Finite Automata

- High-level sketch

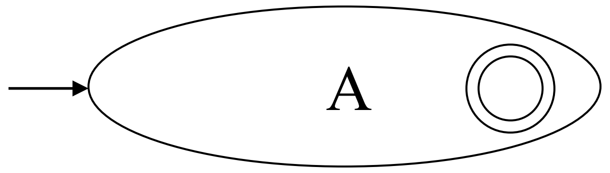


Thompson’s construction

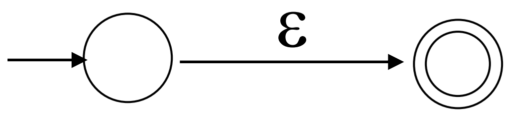
- NFA pattern for each symbol & each operator
- Join them with ε transition in its order appear in the original regular expression.
- ε -NFAs adds a convenient feature but (in a sense) they bring us nothing new: they do not extend the class of languages that can be represented.
- Both NFAs and ε-NFAs recognize the same languages.
- It just help to make converting RE to NFA more convenient and simpler.

RE to ε-NFA using Thompson's construction

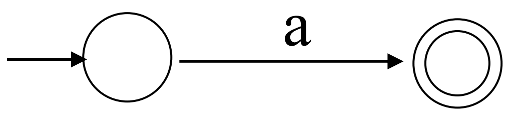
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

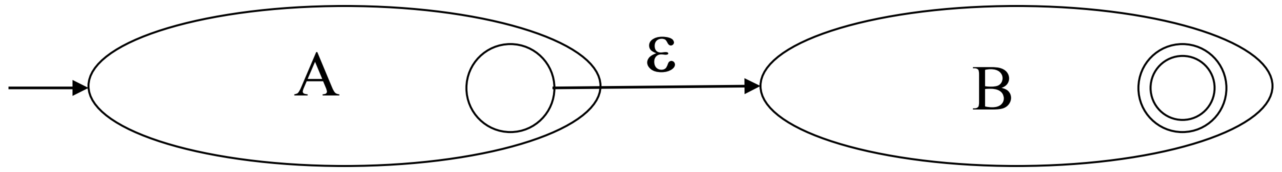


- For input a

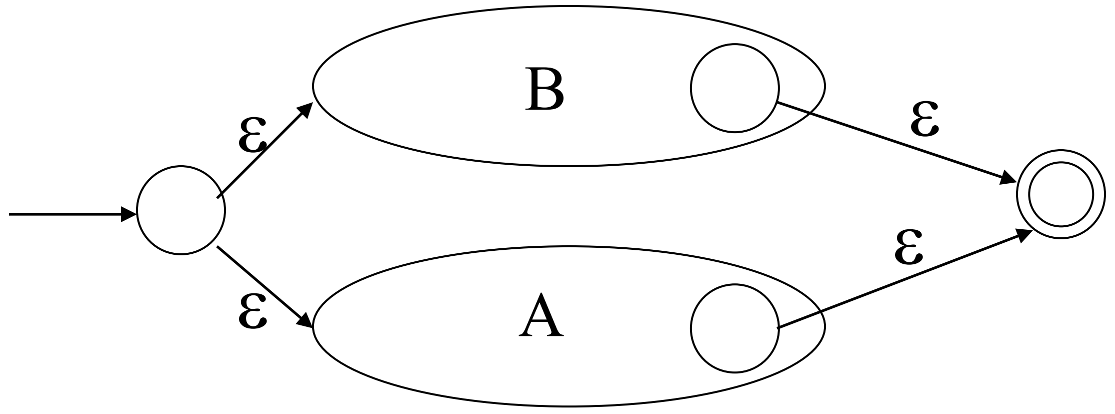


RE to ε-NFA using Thompson's construction

- For AB

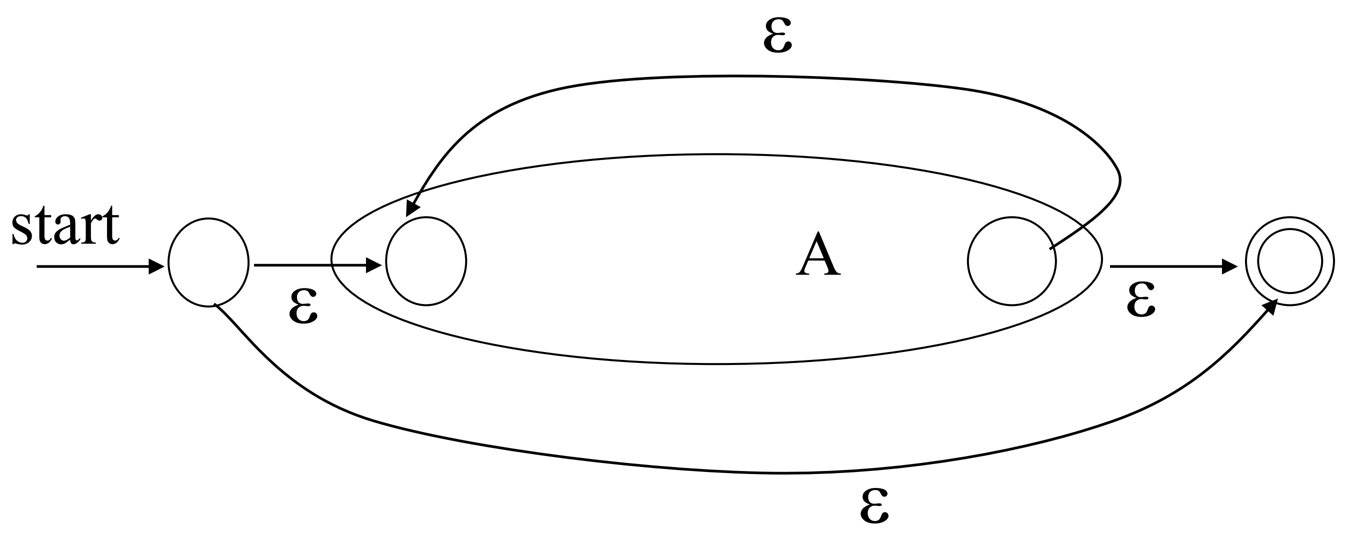


- For A | B



RE to ε-NFA using Thompson's construction

- For A*



Example 1

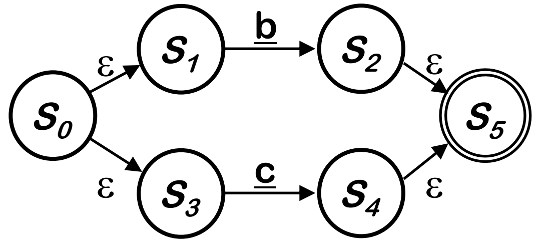
- Build NFA for **a (b | c)***
- **First create a state and translation for each symbol**

$$\begin{tikzcd}
s_0 \arrow[r, "\underline{a}"] & s_1 & s_0 \arrow[r, "\underline{b}"] & s_1 & s_0 \arrow[r, "\underline{c}"] & s_1
\end{tikzcd}$$

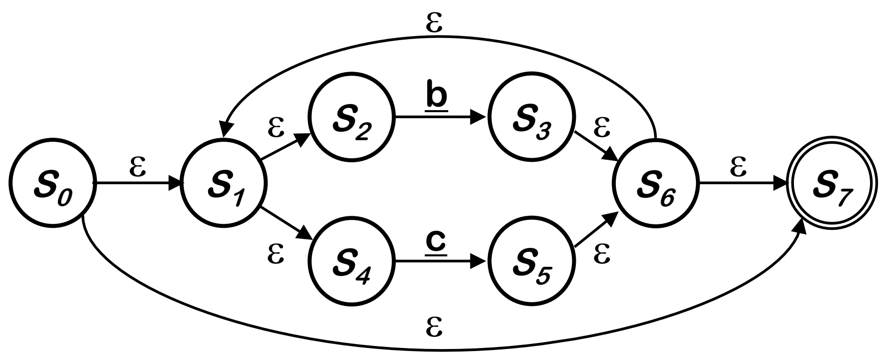
- Now connect them using the magic symbol ε

Example 1

- NFA for (b | c)

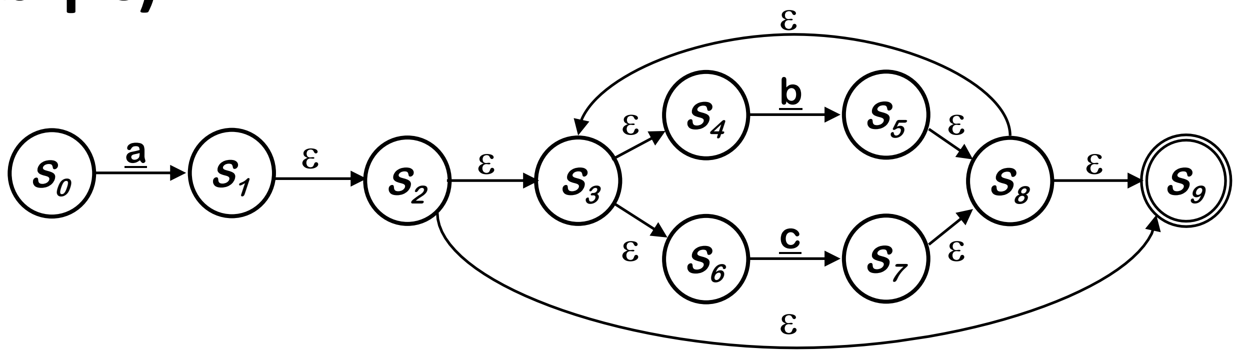


- NFA for (b | c)*

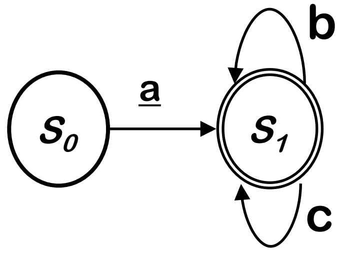


Example 1

- NFA for **a(b | c)***



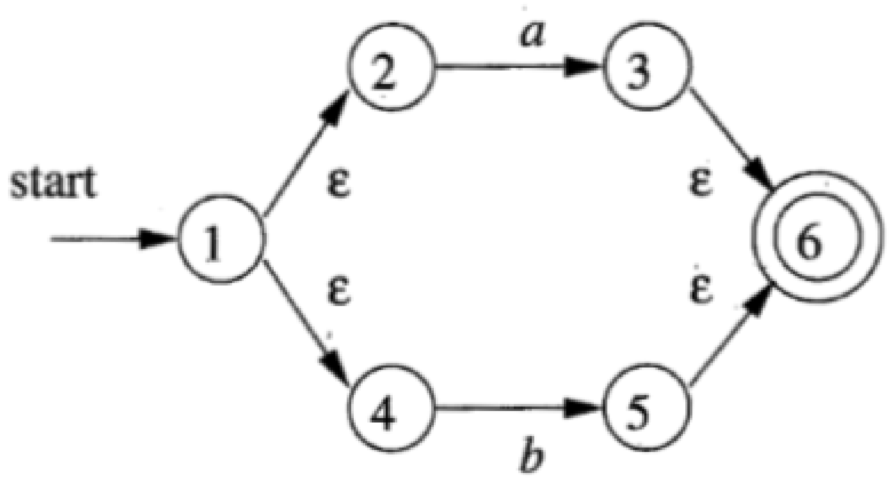
- Of course, a human would design something simpler like →



- But we can automate the production of the more complex NFA version using Thompson's construction, by just constructing smaller NFA patterns then joining them using ε transition

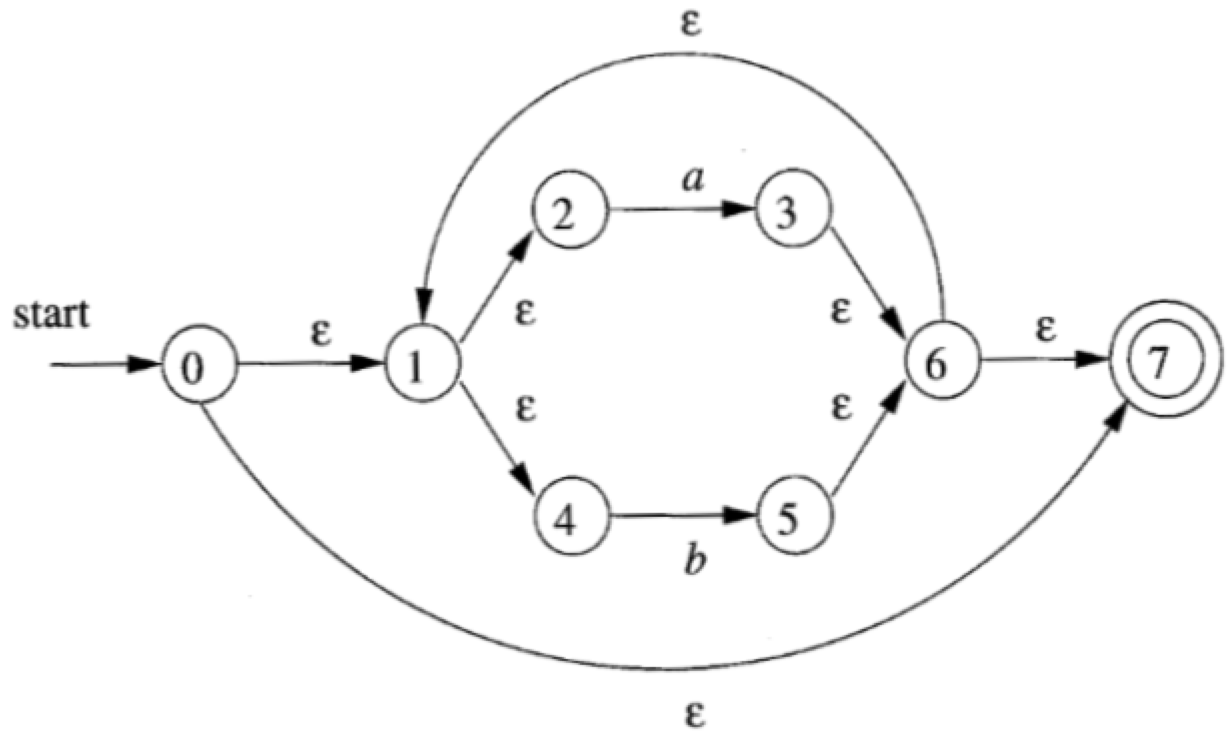
Example 2

- Construct the NFS for the regular expression **(a l b)*abb**
- First we get the NFA for (**a l b)**



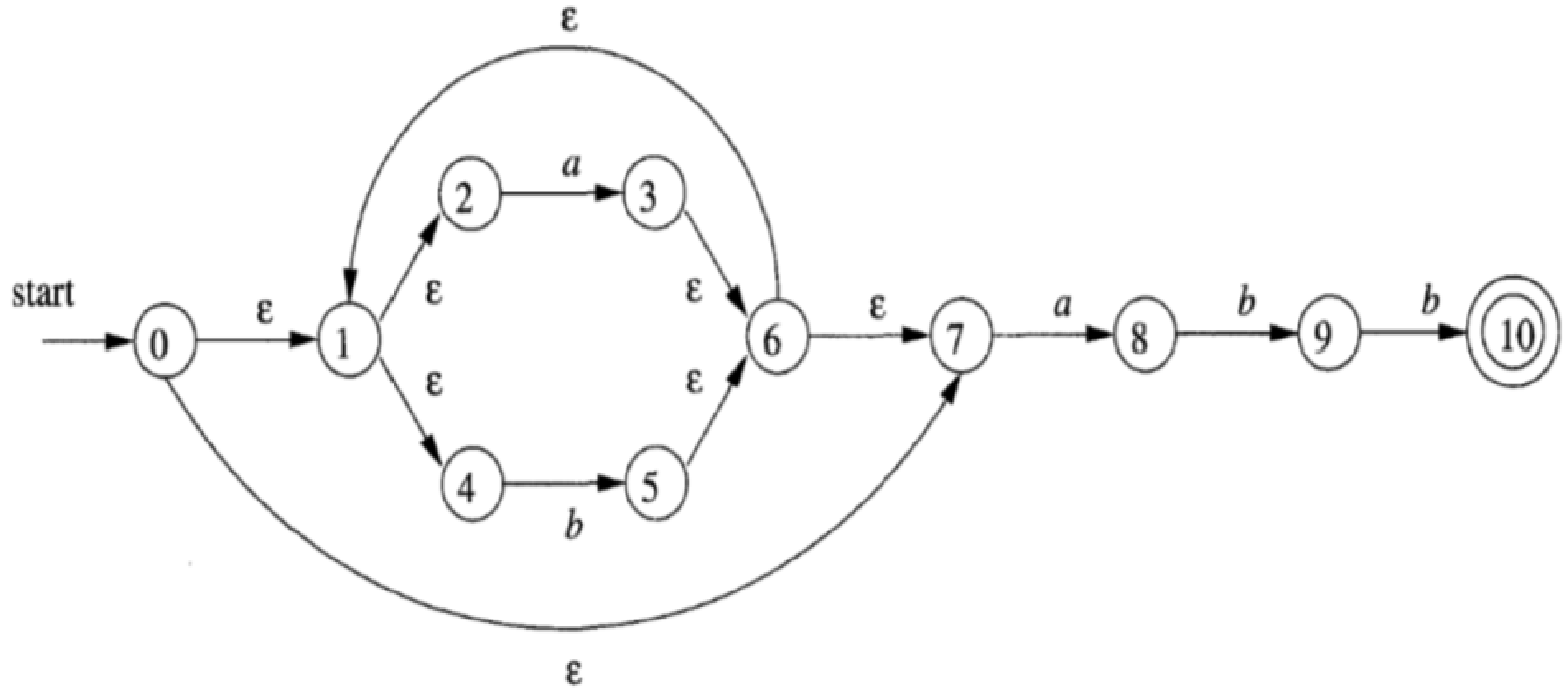
Example 2

- Then we get the NFA for **(a l b)***



Example 2

- Then get the NFA for **(alb)*abb**
-

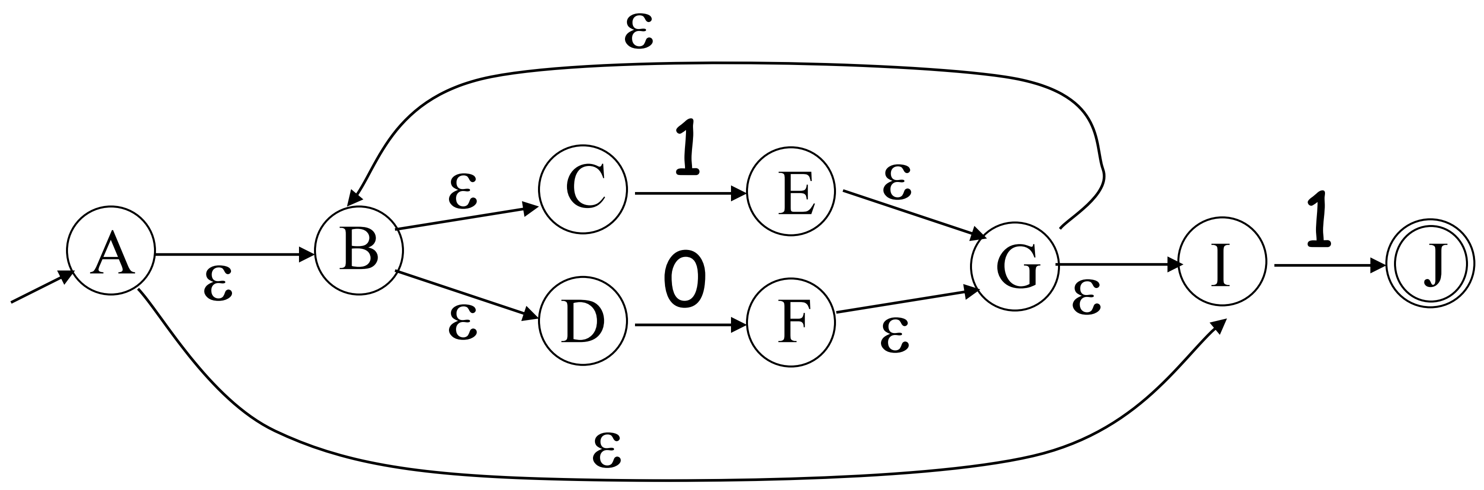


Example 3

- Consider the regular expression

  (1 | 0)*1

- The NFA is



References

- Compilers – Principles, Techniques and Tools, Second Edition by Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman.
- Compiler Construction. Principles And Practice Kenneth C.
- Keith D. Cooper & Linda Torczon ,Cooper: Engineering a Compiler, 2e Lecture Slides (elsevier.com),2010

THANKS
SEE U NEXT LECTURE