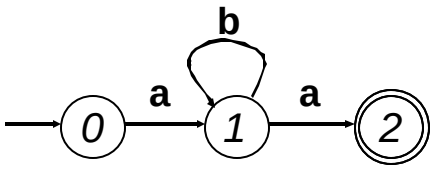
Lecture 7

Table Encoding of FA

A FA can be encoded as a table. This is called a *transition table*. The following example shows a FA encoded as a table.



	a	b
0	*1*	err
1	*2*	*1*
2	err	err

The rows correspond to states. The characters of the alphabet set Σ appear in columns. The cells of the table contain the next state. This encoding makes the implementation of the FA simple and efficient. It is equally simple to simulate or run the FA given an alphabet and a string of the language and its associated alphabet set Σ. The C++ code shows such a FA simulator.

```
int trans_table[NSTATES][NCHARS];
int accept_states[NSTATES];
int state = INITIAL;
while(state != err){
     c = input.read();
     if(c == EOF ) break;
     state=trans_table[state][c];
}
return accept_states[state];
```

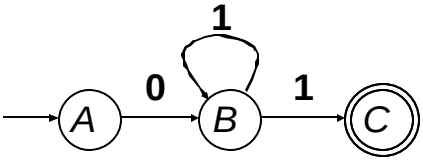
RE ? Finite Automata

We now have a strategy for building lexical analyzer. The tokens we want to recognize are encoded using regular expressions. If we can build a FA for regular expressions, we have our lexical analyzer. The question is can we build a finite automaton for every regular expression? The answer, fortunately, is yes – build FA inductively based on the definition of Regular Expression (RE).

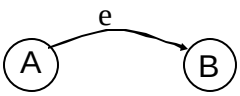
The actual algorithm actually builds N*ondeterministic Finite Automaton* (NFA) from RE. Each RE is converted into an NFA. NFA are joined together with ε-moves. The eventual NFA is then converted into a *Deterministic Finite Automaton* (DFA) which can be encoded as a transition table. Let us discuss how this happens.

Nondeterministic Finite Automaton (NFA)

An NFA can have multiple transitions for one input in a given state. In the following NFA, and input of 1 can cause the automaton to go to state B or C.



It can also have e–moves; the automaton machine can move from state A to state B without consuming input.



The operation of the automaton is not completely defined by input. A NFA can choose whether to make ε-moves and which of multiple transitions to take for a single input. The acceptance of NFA for a given string is achieved if it *can* get in a final state.

Deterministic Finite Automaton (NFA)

In Deterministic Finite Automata (DFA), on the other hand, there is only one transition per input per state. There are no ε-moves. Upon execution of the automaton, a DFA can take *only one path* through the state graph and is therefore completely determined by input.

NFAs and DFAs recognize the same set of languages (regular languages). DFAs are easier to implement – table driven. For a given language, the NFA can be simpler than the DFA. DFA can be exponentially larger than NFA. NFAs are the key to automating RE ? DFA construction.

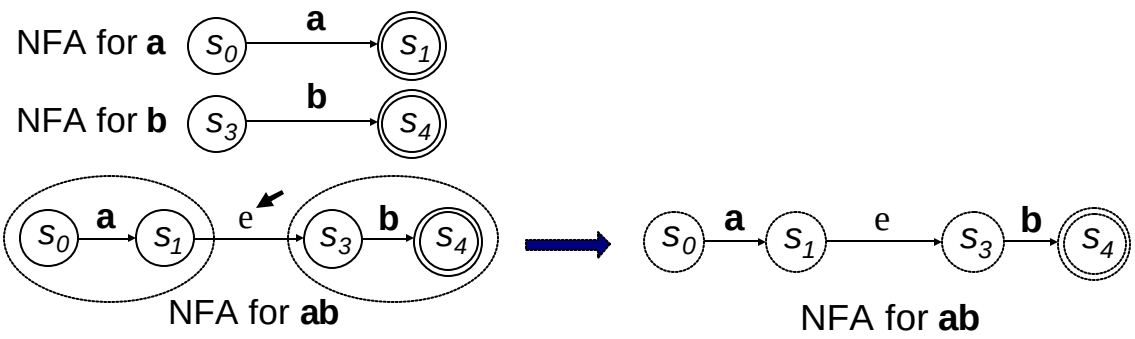
RE ? NFA Construction

The algorithm for RE to DFA conversion is called *Thompson's Construction*. The algorithm appeared in *CACM* 1968. The algorithm builds an NFA for each RE term. The NFAs are then combined using ε-moves. The NFA is converted into a DFA using the subset construction procedure. The number of states in the resulting DFA are minimized using the *Hopcroft's algorithm*.

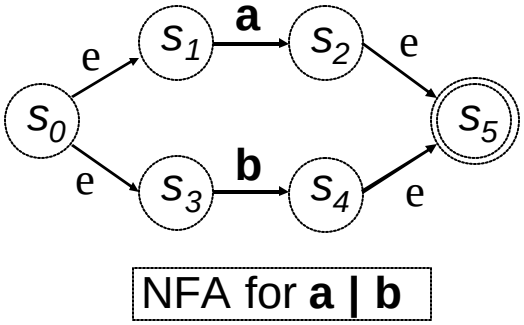
Given a RE, we first create NFA pattern for each symbol and each operator. We then join them with ε-moves in precedence order. Here are examples of such constructions:

1. NFA for RE **ab.**

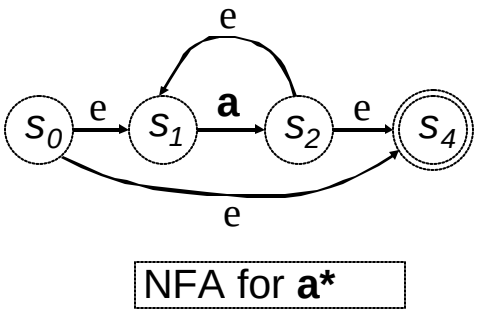
The following figures show the construction steps. NFAs for RE **a** and RE **b** are made. These two are combined using an ε-move; the NFA for RE **a** appears on the left and is the source of the ε-transition.



2. NFA for RE **a|b**



3. NFA for RE **a***



3. NFA for RE **a (b|c)***