Topic 8 – Finite Automata

Key Ideas

- deterministic finite automata (DFA)
- states, start state, accepting states, transitions
- non-deterministic finite automata (NFA)
- ε-non-deterministic finite automata (ε-NFA) and ε-transitions
- transducers

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Deterministic Finite Automata (DFA)

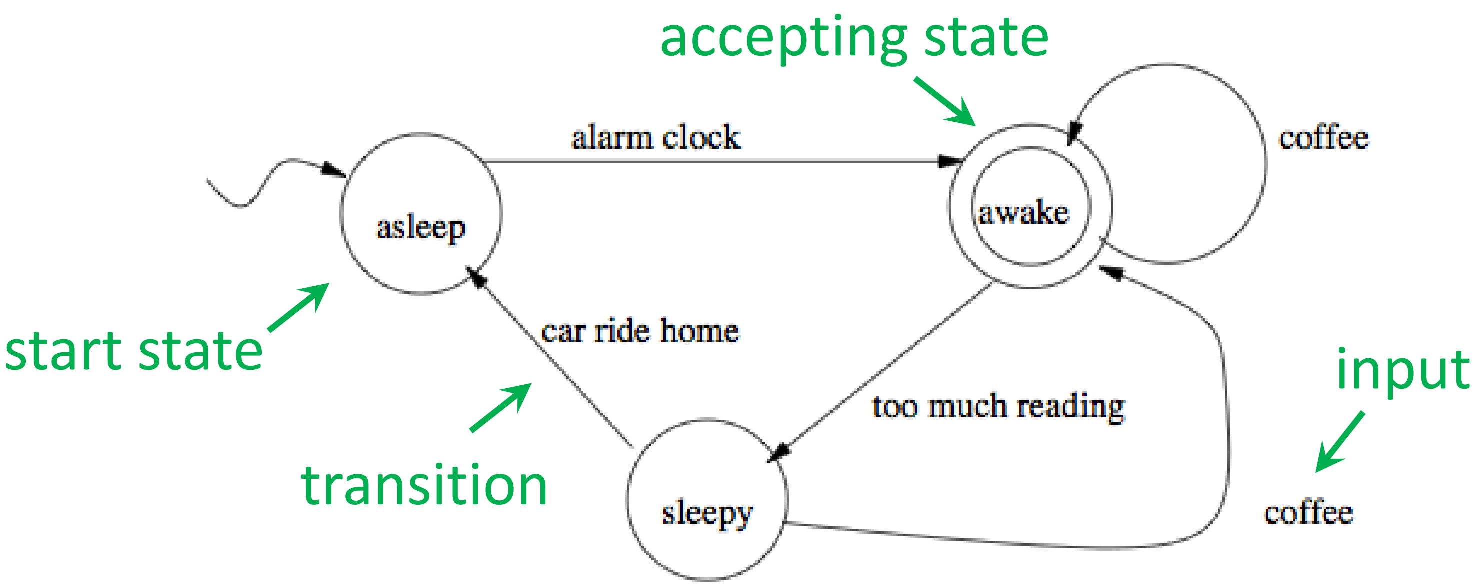
Components

- Also known as a deterministic *finite state machine* (FSM)
- Comprised of
  - A finite *set of states* including
    - one *start state* and
    - at least one (and possibly many) *accepting states*
  - A finite *set of input symbols* known as the alphabet
  - A finite *set of transitions* from one state to another determined by the input
- The DFA determines if input is accepted (a word in the language) or rejected (not in the language)

DFA Picture

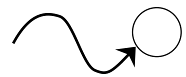
Example

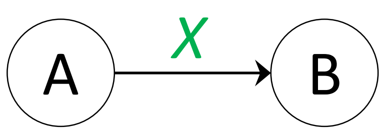
- Start state: asleep (has curvy arrow pointing to it)
- Accepting state, a.k.a. end state: awake (has a double circle)
- Transitions: connect one state (e.g. sleepy) to another state (e.g. awake) based on the value of the input (e.g. coffee)



Deterministic Finite Automata (DFA)

Components of a DFA

- States: circles ○
  - start state: curved line 
  - accepting state(s): two concentric circles ◎
  - may have a label inside (useful but not needed)
- Transitions
  - an edge that moves from one state to another
  - labelled with an input, say *X*

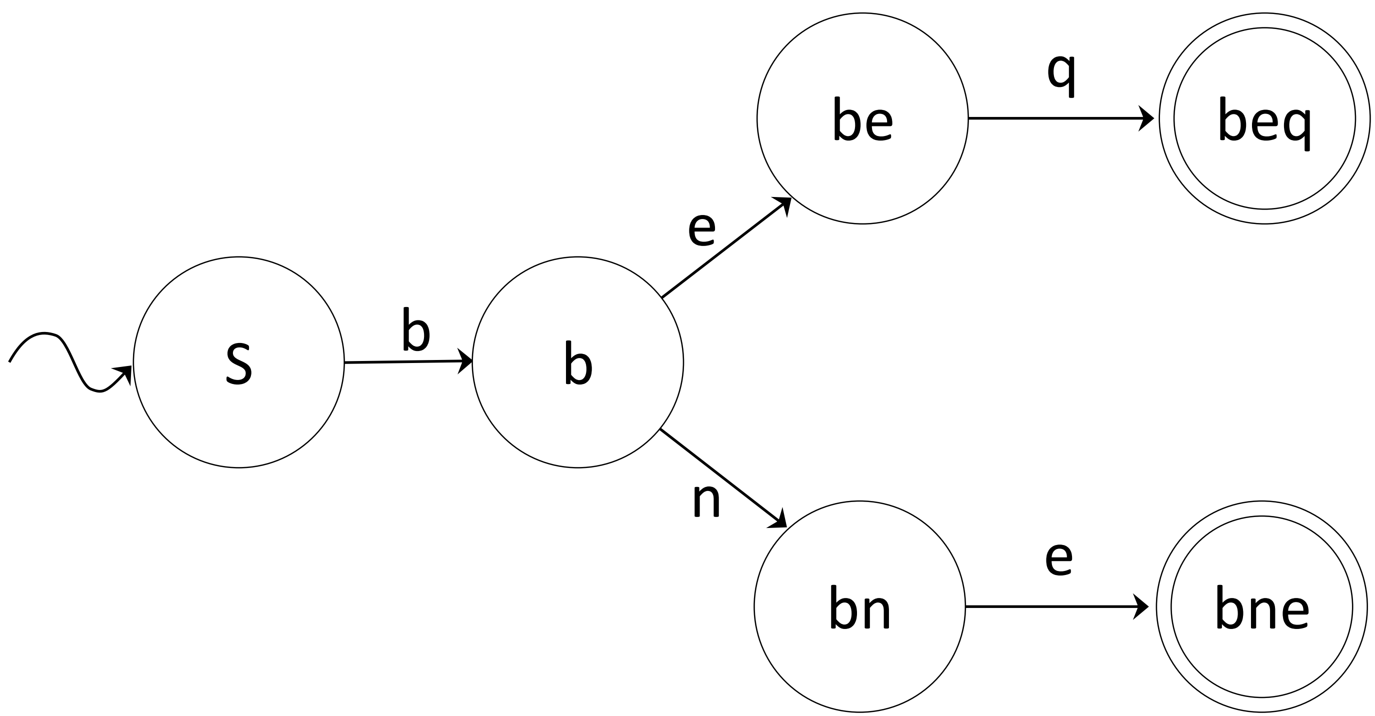


  - it means: on input *X*, move from state A to state B.

Deterministic Finite Automata (DFA)

Example of a DFA that Accepts a Finite Language

- Create a DFA that recognizes the MIPS branch instructions, i.e Σ ={b,e,n,q} and $\mathcal{L}$ = {bne, beq}



Parts of a DFA

Comparison to Programming Languages

Similar to what you would see in a program

- a unique place to start
- transitions to various states and
- one (or possibly many) places to end.

Start State → `int main () {`

Transition → `if (x > 0) {`

End State → `return 0;`

```
int main () {
    ...
    if (x > 0) {
        ...
    }
    return 0;
}
```

Deterministic Finite Automata (DFA)

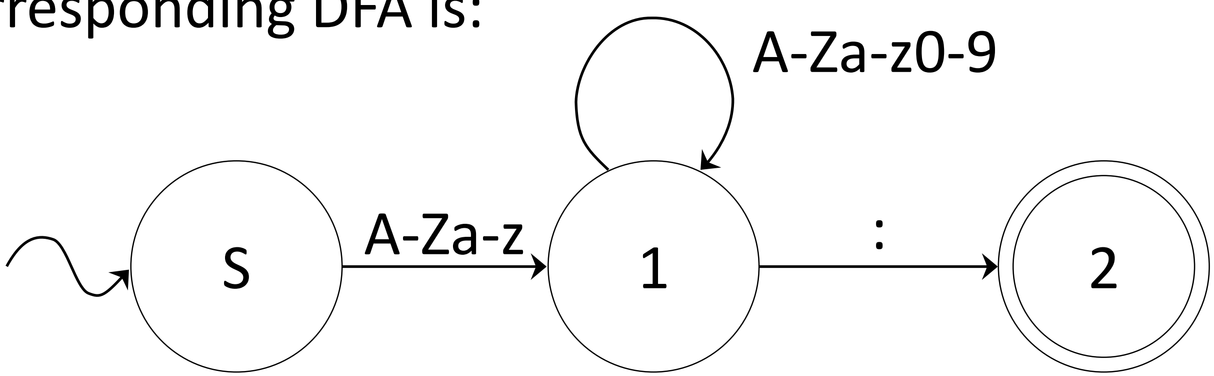
Features of a DFA

- Easy to trace where you are in the computation
- it is *deterministic*, i.e. for each state, the transitions out of that state are uniquely labelled
- *there is no explicit error state*
  - If you are in a state, and the DFA gets an input, say x, such that there is no edge out of that state with that label on it, it is an error.
- The language accepted by the DFA M is called $\mathcal{L}(M)$
  - two slides back $\mathcal{L}(M)$ = {bne, beq}.

Deterministic Finite Automata (DFA)

Example of a DFA that Accepts an Infinite Language

- The regular expression that defines a valid MIPS label definition is $\mathcal{L}$ = [a-zA-Z][a-zA-Z0-9]*:
  - it starts with a letter (capital or small)
  - followed by letters or numbers
  - ends with a colon
- Here we use a-z to refer to all the small letters and 0-9 to refer to all the single digit numbers.
- The corresponding DFA is:



Deterministic Finite Automata (DFA)

Examples of DFAs

Let Σ ={a,b,c}

- Exercise 1: Create a regular expression and a DFA that accepts the language of strings that contain exactly one *a*, one *b*, and no *c*'s.

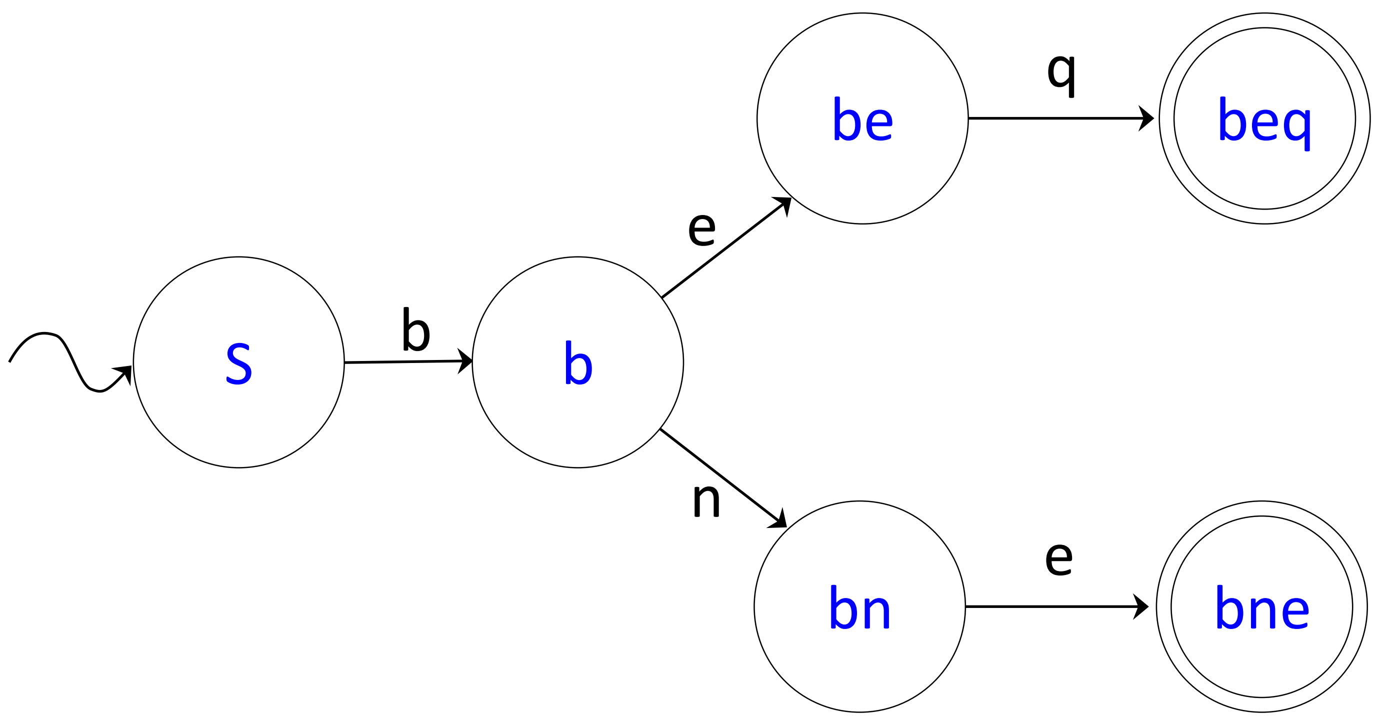
- Exercise 2: Create a regular expression and a DFA that accepts the language of strings that contain at least one *a*.

- Exercise 3: Create a regular expression and a DFA that accepts the language of strings that contain an even number of *a*'s (including 0 *a*'s).

Deterministic Finite Automata (DFA)

Recall this Example of a DFA

- This DFA recognizes the MIPS branch instructions, i.e. $\Sigma = \{b,e,n,q\}$ and $\mathcal{L} = \{bne, beq\}$



Deterministic Finite Automata (DFA)

Formal Definition

A DFA is a 5-tuple (Σ , Q, q_0, A, δ) where

- Σ is a finite alphabet, e.g. Σ ={b,e,n,q}
- Q is a finite set of states, e.g. Q={S, b, be, bn, beq, bne}
- q_0 is start state, e.g. q_0 = {S}
- A is the set of accepting states, e.g. A= { beq, bne }
- δ: Q x $\Sigma \rightarrow \Sigma$ is a transition function that maps from the set of (state, symbol) pairs to a state, e.g. δ(S, b) = b; δ(b, e) = be; δ(b, n) = bn; δ(be, q) = beq; δ(bn, e) = bne.

  E.g. δ(*b*, *e*) = *be* means if the DFA is in state *b* and the input is *e*, then go to state *be*.

Deterministic Finite Automata (DFA)

Implementing a DFA

- Input, a sequence of characters from Σ: $c_1, c_2, \ldots c_n$

state $\leftarrow q_0$ *// start in the start state*

for i = 1 to n **do**:   *// for each character in the input*

    state $\leftarrow \delta$ (state, c_i)   *// change state based on the input*

return (state $\in$ A)   *// did it end in an accepting state*

- Ouput TRUE means $c_1, c_2, \ldots c_n$ is a word in the language accepted by the DFA, output FALSE otherwise.
- Implement δ (state, c_i) as a table…

Deterministic Finite Automata (DFA)

Implementing a DFA

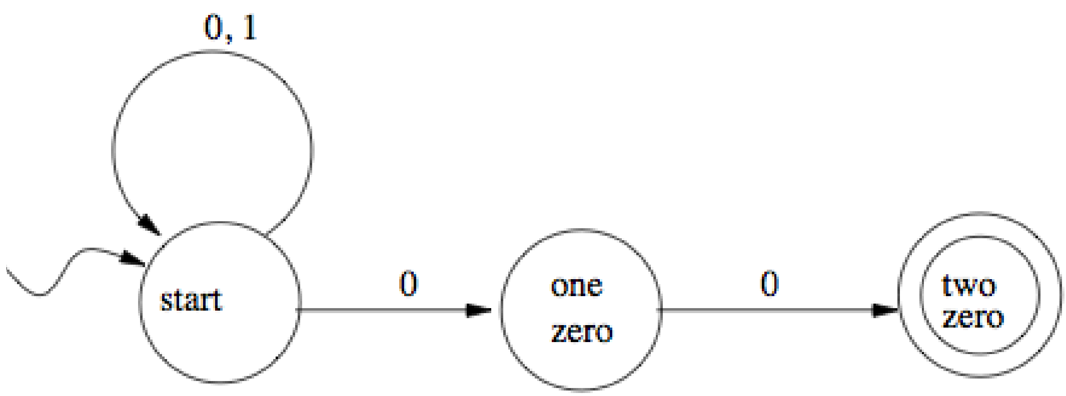
- Implement δ as a table were each row corresponds to a different state, each column to a letter in the alphabet, Σ, and 🚫 means error.

δ	b	e	n	q
S	b	🚫	🚫	🚫
b	🚫	be	bn	🚫
bn	🚫	bne	🚫	🚫
bne	🚫	🚫	🚫	🚫
be	🚫	🚫	🚫	beq
beq	🚫	🚫	🚫	🚫

Non-deterministic Finite Automata (NFA)

How a NFA Differs

- Key Difference: In a NFA, *two or more edges leaving the same state can have the same label and lead to different states.*
- The next state in non-deterministic, i.e. a set of possible states rather than a single state.
- In state *start,* with input 0, the NFA can stay in *start* or go to state *one zero,* i.e. it's next state is the set *{start, one zero}.*



Non-deterministic Finite Automata (NFA)

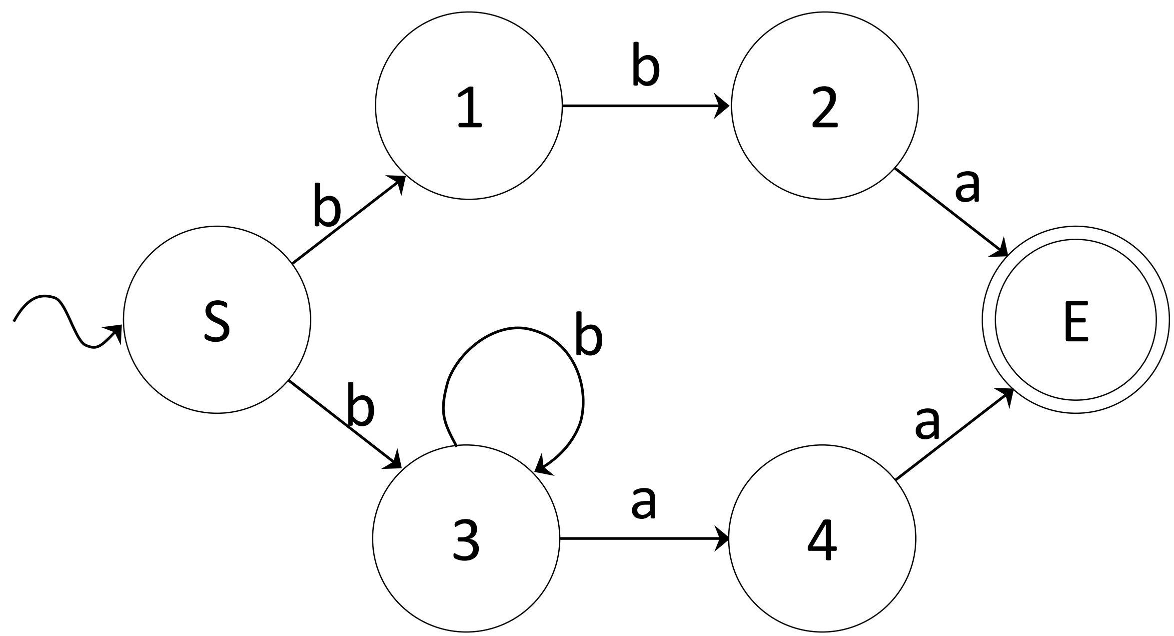
Comparison with DFA

- A language is accepted if *at least one path* leads to an accepting state.
- A language is rejected if *no path* leads to an accepting state.
- The NFA on the previous slide accepts the language of words that end with ‘00’.
- It is often easier to design an NFA rather than an equivalent—but more complex—DFA (e.g. to tokenize input).
- Algorithms exist to convert an NFA to an equivalent DFA.

Non-deterministic Finite Automata (NFA)

Comparison with DFA

- Let Σ ={a, b} and let $\mathcal{L}$ = {bba, bb*aa}, i.e. $\mathcal{L}$ is: 2 *b*'s followed by an *a* or at least one *b* followed by two *a*'s.
- First try this as a DFA.
- Next consider the NFA:
- If we are in state *S* and we get input *b* *we move to the set of states {1, 3}.*



Non-deterministic Finite Automata (NFA)

Comparison with DFA

- An NFA is a FA that *allows you to be in multiple states at the same time,* i.e. a set of states.
- Terminology: 2^Q is the *power set* of Q, i.e. all the possible subsets of Q.
- E.g. if Q = {a, b, c} then 2^Q is

  { { }, {a}, {b}, {c}, {ab}, {ac} {bc}, {abc} }
- We use the notation 2^Q because $| 2^Q | = 2^{|Q|}$
- For a NFA the *transition relation maps onto a set of states rather than a single state*, T: $Q \times \Sigma \rightarrow 2^Q$

Non-deterministic Finite Automata (NFA)

Implementing a NFA

- Input, a sequence of characters from Σ: $c_{1,}$ c_2, … c_n

states $\leftarrow q_0$ *// start in the start state*
for each c_i in input **do**: *// for each char in input*
    s' = { }
    for each s in states **do:** *// for each state you are in,*
        s' = s' U T(s, c_i) *// find possible next states*
    states $\leftarrow$ s'
return (states $\cap$ A $\neq$ {}) *// check if final state is accepting*

- Output TRUE if one of the states you end up in is an accepting state (i.e. in the set A)

Non-deterministic Finite Automata (NFA)

Comparison with DFA

- Let Σ ={a, b} and let $\mathcal{L}$ = {(a|b)*bbb(a|b)*}, i.e. $\mathcal{L}$ is the set of words with three b's in a row.
- NFA version
- DFA version
- What about 4 b's in a row? 5 b's? 6 b's?

Working with DFAs vs. NFAs

DFAs

- *easier:* to implement

NFAs

- *simplier:* tend to have less states than a corresponding DFA for that accepts the same language
- *slower:* require a set data type

- The two types have the same expressive power.
- I.e. languages that can be identify with one, can be identify with the other.

Deterministic Finite Automata (DFA)

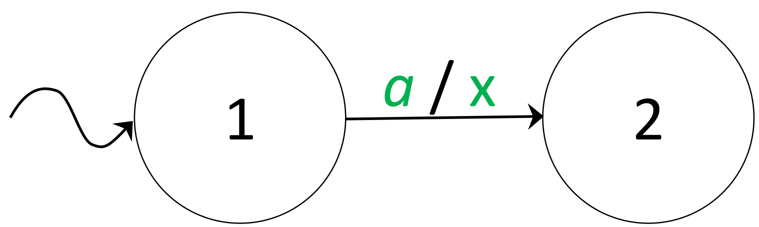
Where are DFA's used?

- lexer / scanner / translating
- transforming input (transducers)
- searching in text
- a computer processor is a highly complex DFA where
    - the states are the values of all the registers and the stack
    - the input is the next instruction (fetched from RAM)
- Alan Turing imagined a computer as a combination of a finite state machine + memory
    - in his case a memory = tape
    - now we use RAM

Extensions

Transducers

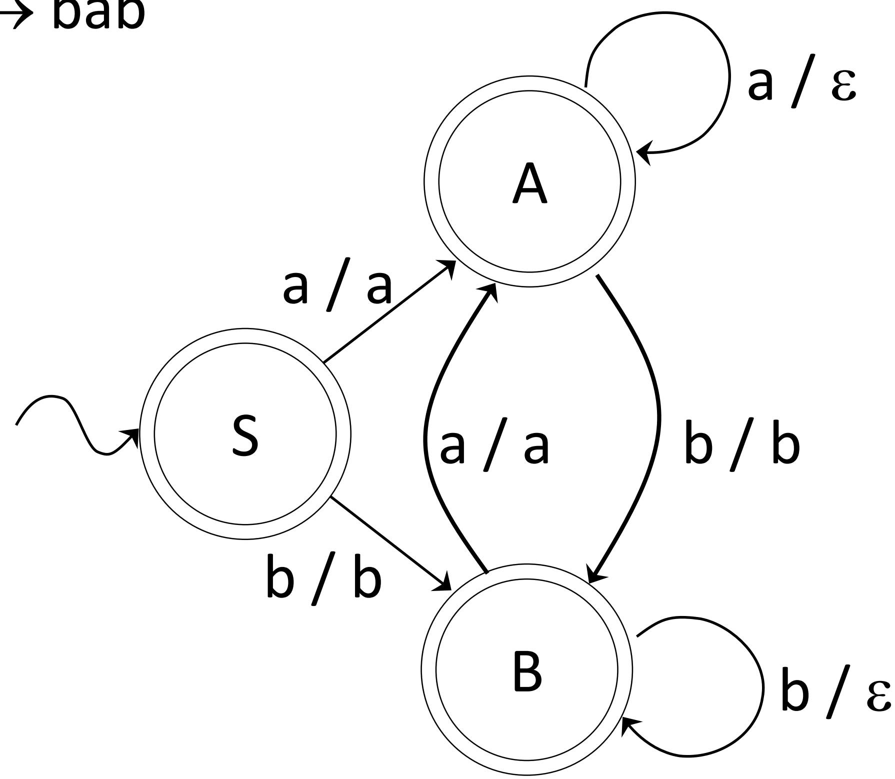
- *extension*: for each transition, provide the ability to output a single character
- e.g. if the FA is in state 1, and the next input character is an *a*, then output an *x* and go to state 2.



Extensions

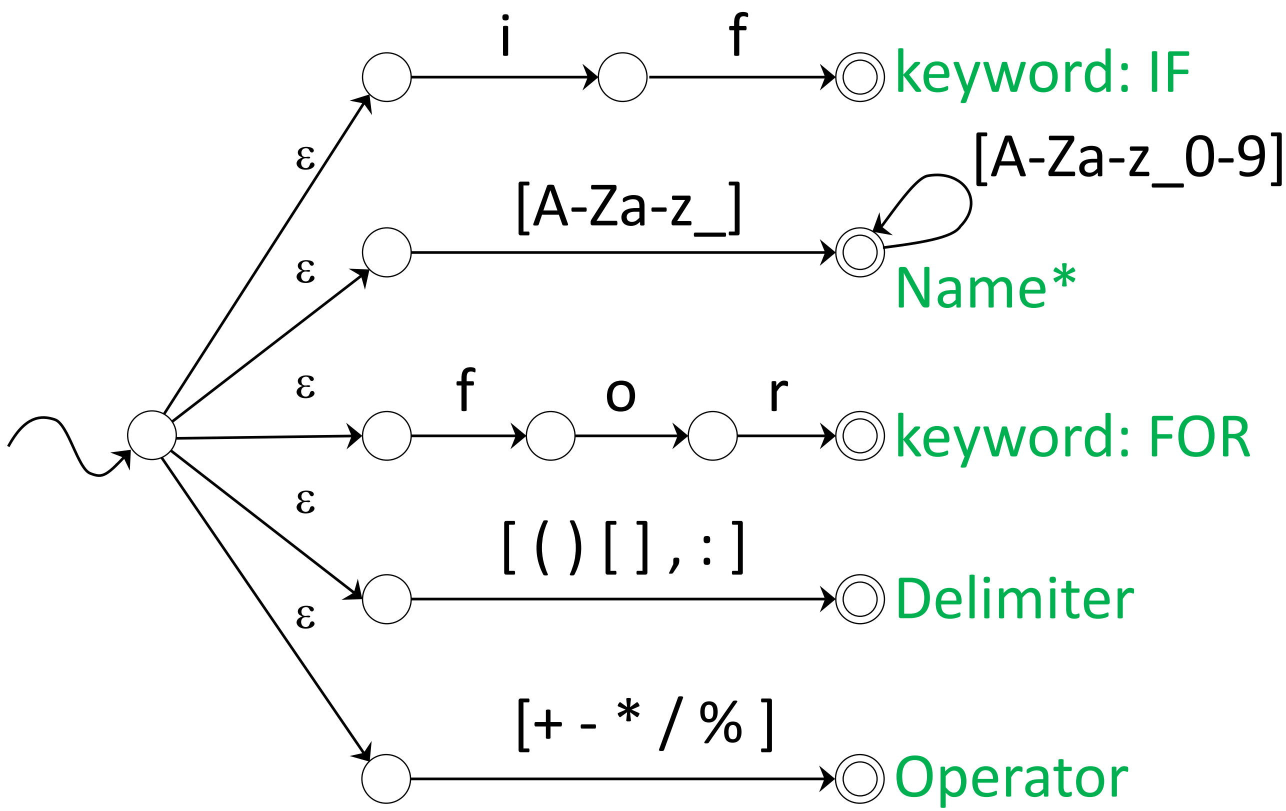
Transducers

- This transducer removes stutters (the same character more than once in a row) from the input stream, i.e. aaabbaa $\rightarrow$ aba baaaaaabbb $\rightarrow$ bab



ε-Non-deterministic Finite Automata (ε-NFA)

- An *ε-NFA* allows the use of *ε-transitions* (i.e. a transition that happens without consuming any input).

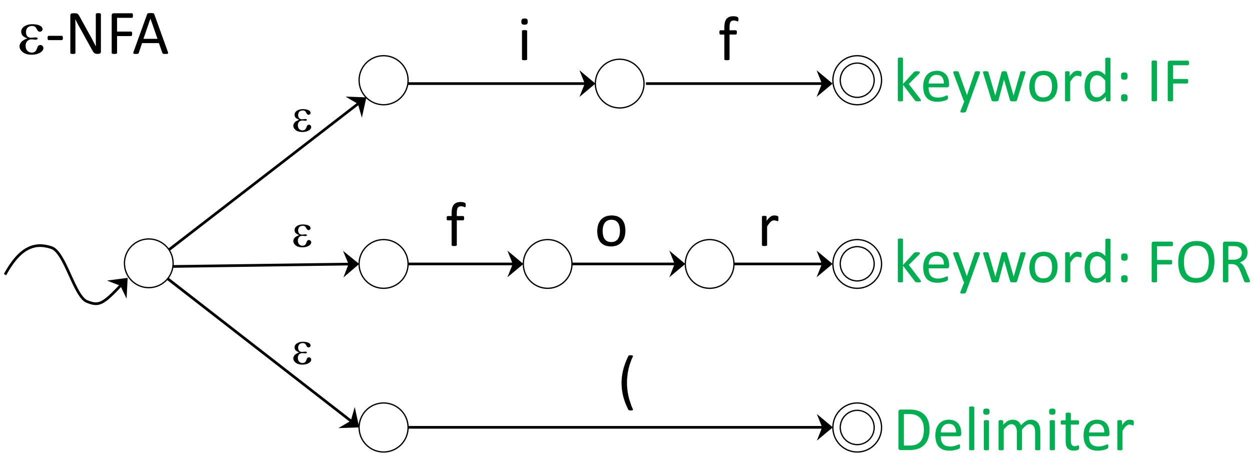


*A lexar must check that a potential name is not a keyword.

ε-Non-deterministic Finite Automata (ε-NFA)

ε-Transitions

- allow transitions from one state to another without consuming (or requiring) any input
- makes it easy to join different FA's together
- easy to convert an ε-NFA to an NFA



ε-Non-deterministic Finite Automata (ε-NFA)

ε-Transitions

- Equivalent NFA