# Compiler Construction
# Lent Term 2020
# Lexical analysis
# Lecture 13

**Timothy G. Griffin**
**tgg22@cam.ac.uk**
**Computer Laboratory**
**University of Cambridge**

# What problem are we solving?

Translate a sequence of characters

```
if m = 0 then 1 else if m = 1 then 1 else fib (m - 1) +  fib (m -2)
```

into a sequence of **tokens**

```
IF, IDENT “m”, EQUAL, INT 0, THEN, INT 1, ELSE, IF,
IDENT “m”, EQUAL, INT 1, THEN, INT 1, ELSE, IDENT “fib”,
LPAREN, IDENT “m”, SUB, INT 1, RPAREN, ADD,
IDENT “fib”, LPAREN, IDENT “m”, SUB, INT 2, RPAREN
```

implemented with some data type

```
type token =
    | INT of int| IDENT of string | LPAREN | RPAREN
    | ADD | SUB | EQUAL | IF | THEN | ELSE
    | …
```

# Regular expressions $e$ over alphabet $\Sigma$

$$e \to \phi \mid \varepsilon \mid a \mid e+e \mid ee \mid e^* \qquad (a \in \Sigma)$$

$$M(e) \subseteq \Sigma^*$$

$$M(\phi) = \{\}$$

$$M(\varepsilon) = \{\varepsilon\}$$

$$M(a) = \{a\}$$

$$M(e_1 + e_2) = M(e_1) \cup M(e_2)$$

$$M(e_1 e_2) = \{w_1 w_2 \mid w_1 \in M(e_1),\ w_2 \in M(e_2)\}$$

$$M(e^0) = \{\varepsilon\}$$

$$M(e^{n+1}) = M(ee^n)$$

$$M(e^*) = \bigcup_{n \geq 0} M(e^n)$$

# Regular Expression (RE) Examples

$$M((a+b)^{*}abb) =$$

$$\{abb,\ aabb,\ baabb,\ aaabb,\ ababb,$$

$$baabb,\ bbabb,\ aaaabb, \cdots\}$$

$$M((\Xi + \otimes)^{*}\Xi \otimes \otimes) =$$

$$\{\Xi \otimes \otimes,\ \Xi\Xi \otimes \otimes,\ \otimes \Xi\Xi \otimes \otimes,$$

$$\Xi\Xi\Xi \otimes \otimes,\ \Xi \otimes \Xi \otimes \otimes, \otimes\Xi\Xi \otimes \otimes,$$

$$\otimes \otimes \Xi \otimes \otimes,\ \Xi\Xi\Xi\Xi \otimes \otimes, \cdots\}$$

# Review of Finite Automata (FA)

$$M = (Q, \Sigma, \delta, q_0, F)$$

$Q$ : states

$\Sigma$ : alphabet

$q_0 \in \mathrm{Q}$ : start state

$F \subseteq \mathrm{Q}$ : final states

$$\forall q \in Q, a \in \Sigma, \delta(\mathrm{q}, \mathrm{a}) \in \mathrm{Q}$$

for deterministic FA (DFA)

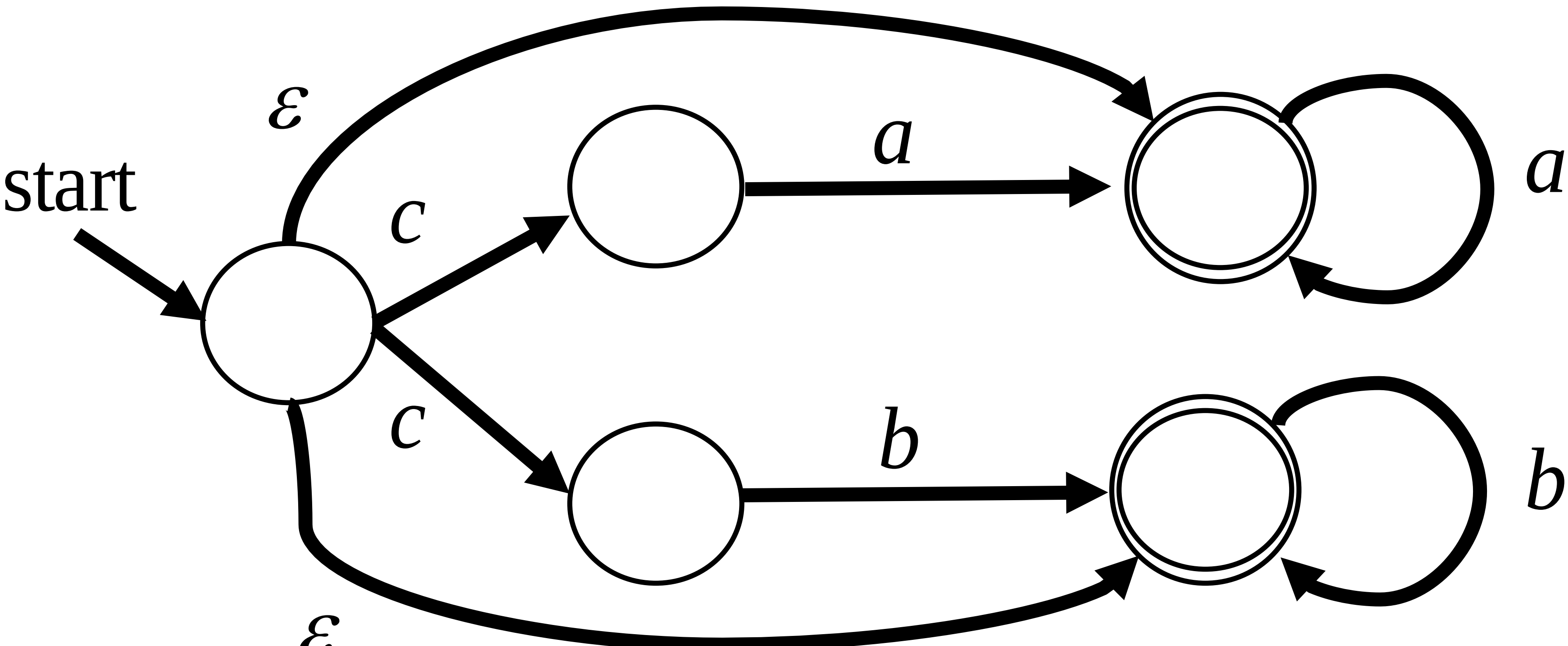
$$\forall q \in Q, a \in (\Sigma \cup \{\varepsilon\}), \delta(\mathrm{q}, \mathrm{a}) \subseteq \mathrm{Q}$$

for nondeterministic FA (NFA)

# NFA Example

An NFA accepting

$$M(a^* + b^* + caa^* + cbb^*)$$



# A bit of notation

For deterministic FA.

$$q \xrightarrow{\varepsilon} q$$

$$q_1 \xrightarrow{aw} q_3 \quad \text{if } \delta(q_1, a) = q_2 \text{ and } q_2 \xrightarrow{w} q_3$$

$$L(M) = \{w \mid \exists q \in F, q_0 \xrightarrow{w} q\}$$

---

For nondeterministic FA.

$$q \xrightarrow{\varepsilon} q$$

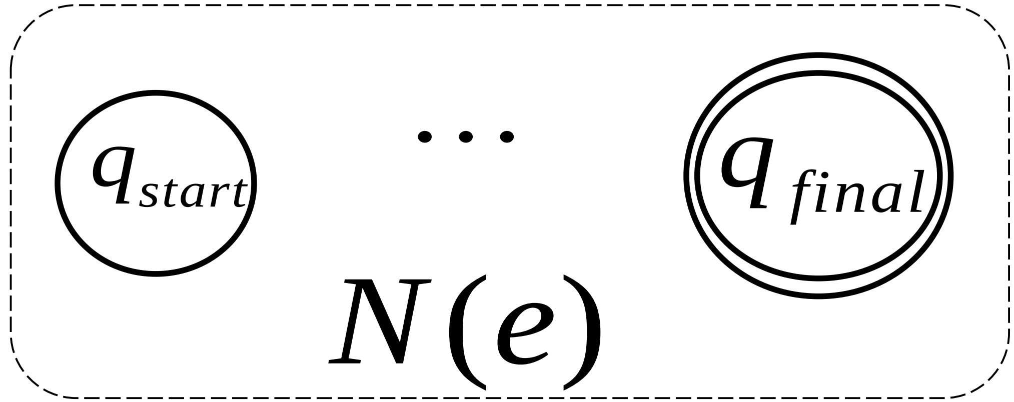
$$q_1 \xrightarrow{w} q_3 \quad \text{if } \quad q_2 \in \delta(q_1, \varepsilon) \quad \text{and } q_2 \xrightarrow{w} q_3$$

$$q_1 \xrightarrow{aw} q_3 \quad \text{if } \mathrm{q}_2 \in \delta(q_1, a) \quad \text{and } q_2 \xrightarrow{w} q_3$$

$$L(M) = \{w \mid \exists q \in F, q_0 \xrightarrow{w} q\}$$

# Review of RE -> NFA

$e \quad \Rightarrow$



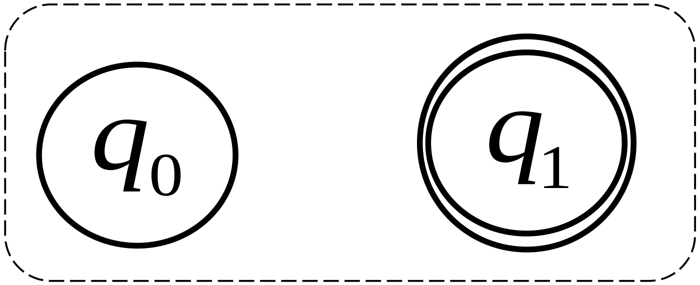

A regular expression.

A nondeterministic FA accepting *M(e) with a single final state.*

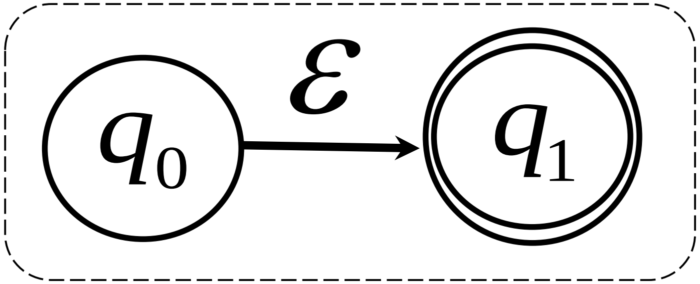
The construction is done by induction on the structure of *e*.

# Review of RE -> NFA

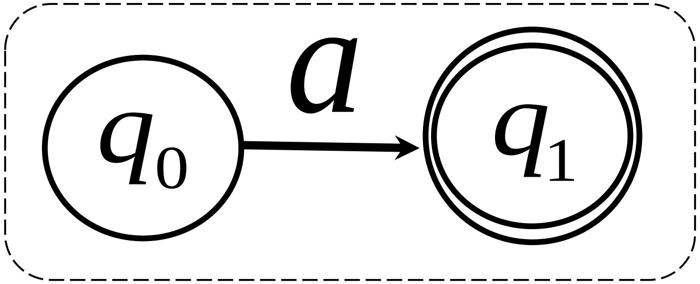
$$N(\phi) =$$




$$N(\varepsilon) =$$

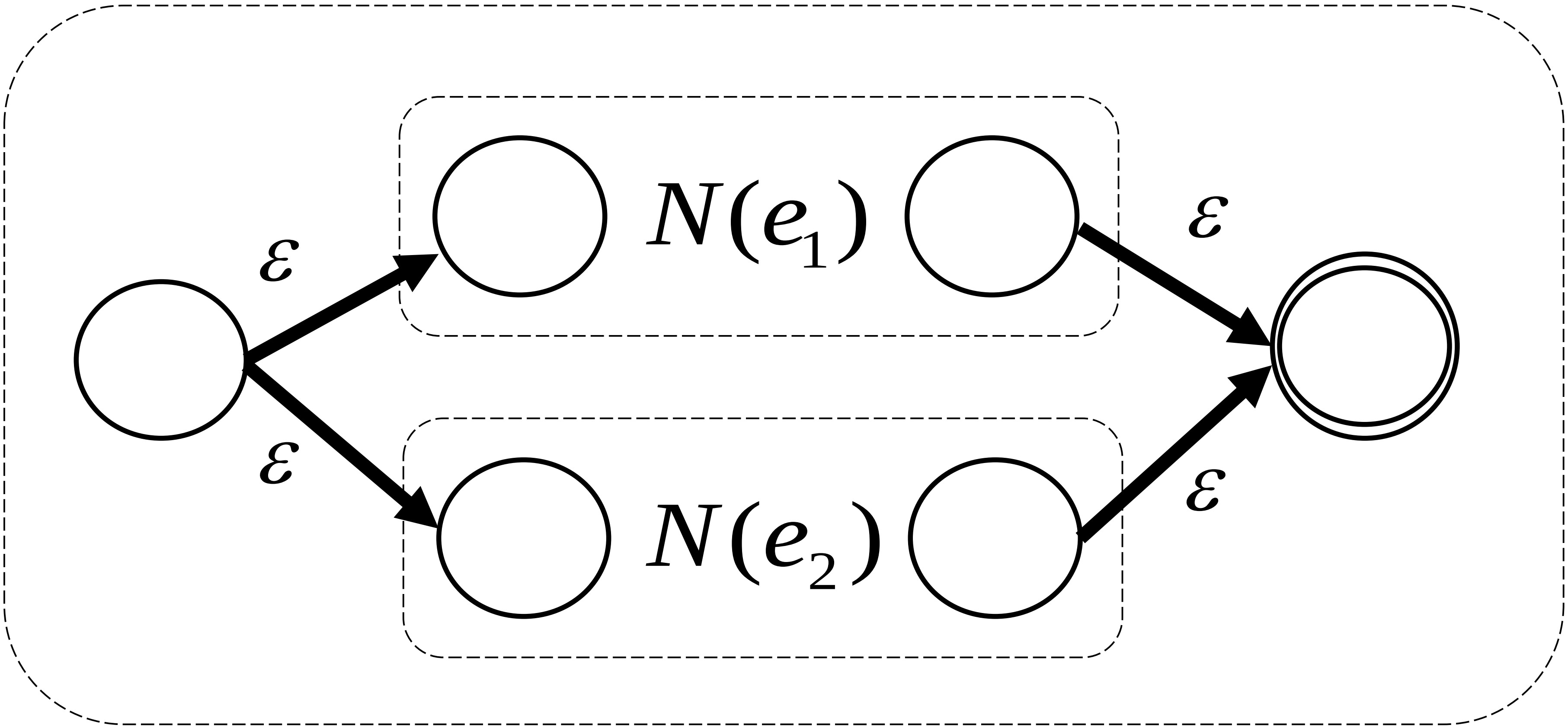



$$N(a) =$$



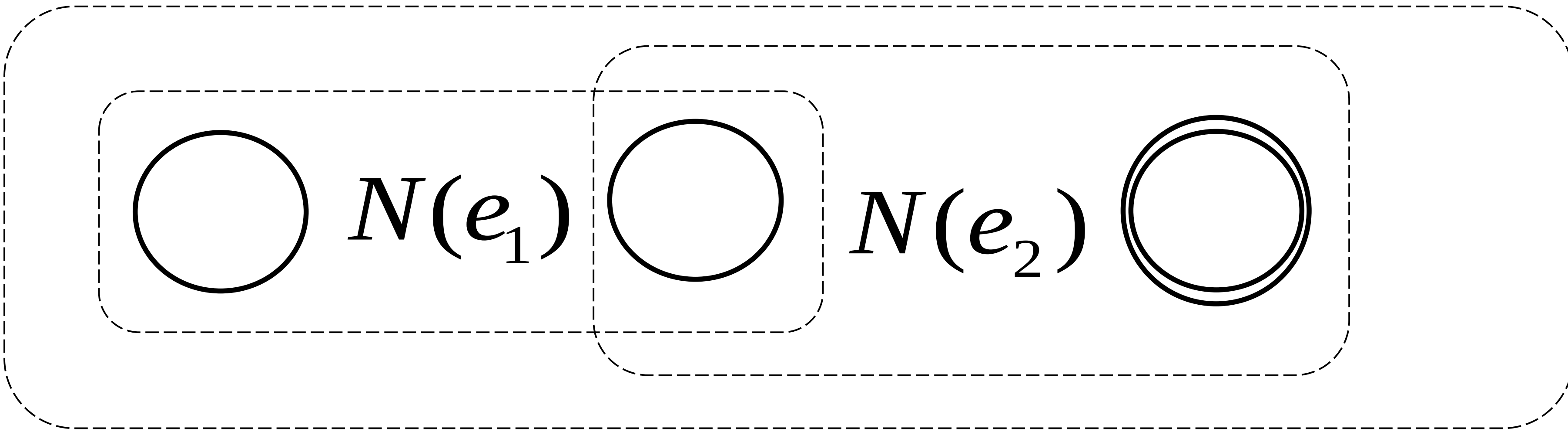
# Review of RE -> NFA

$$N(e_1 + e_2) =$$



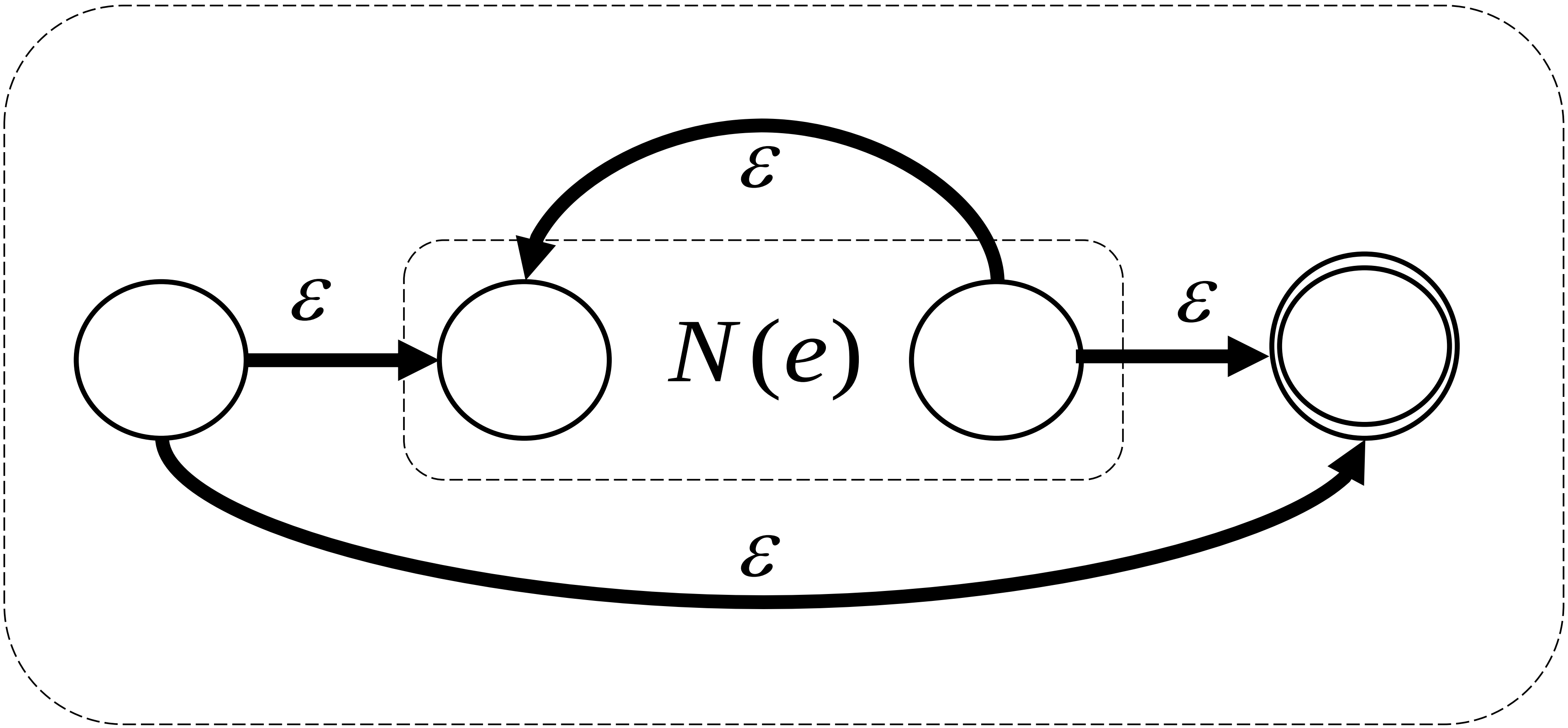
# Review of RE -> NFA

$$N(e_1 e_2) =$$

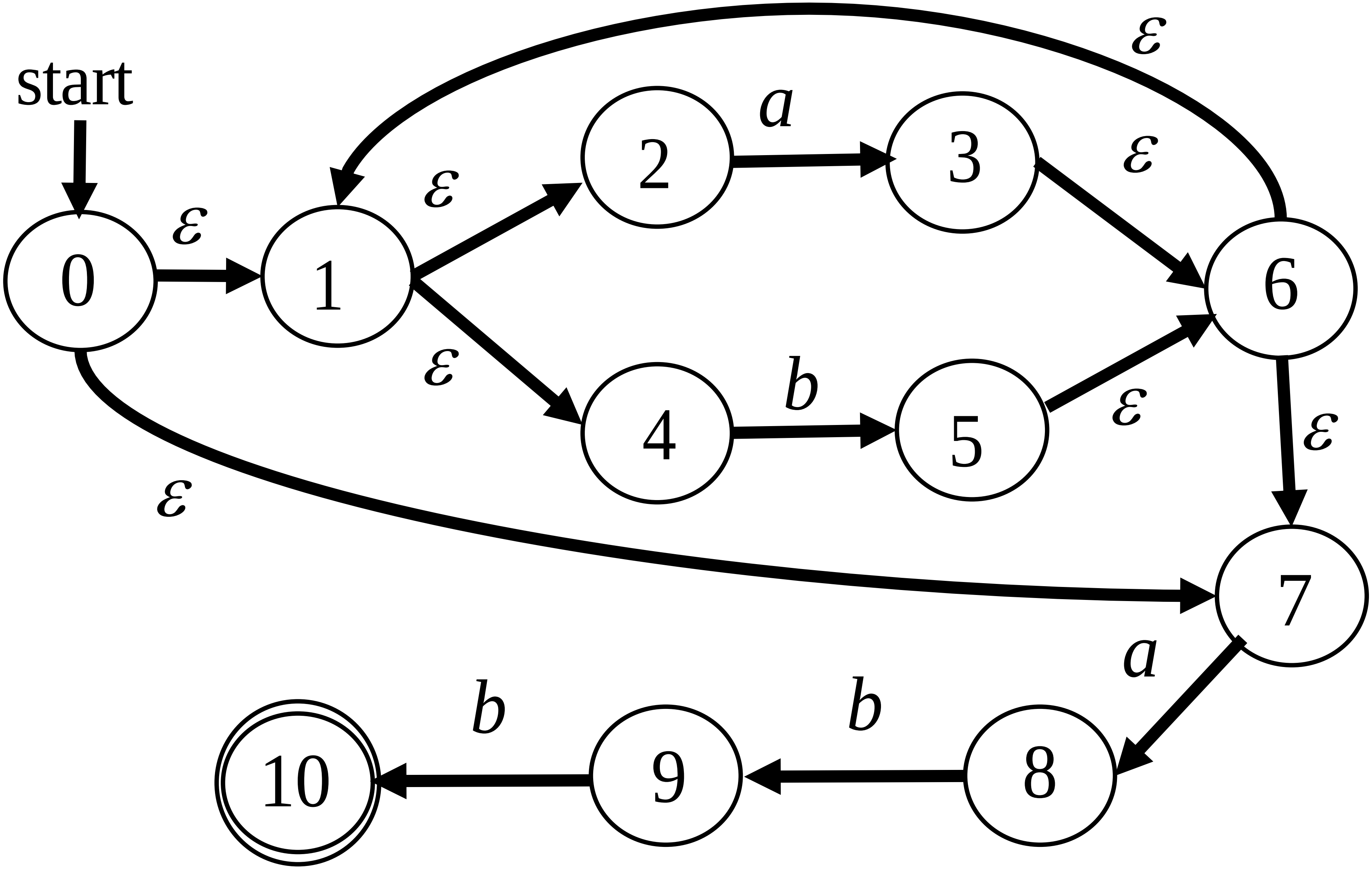


# Review of RE -> NFA

$$N(e^{*}) =$$



# $N((a+b)^* abb)$



# Review of NFA -> DFA

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$M' = (Q', \Sigma, \delta', q_0', F')$$

$$Q' = \{S \mid S \subseteq Q\}$$

$$\varepsilon - closure(S) = \{q' \in Q \mid \exists q \in S, q \xrightarrow{\varepsilon} q'\}$$

$$\delta'(S, a) = \varepsilon - closure(\{q' \in \delta(q, a) \mid q \in S\})$$

$$q_0' = \varepsilon - closure\{q_0\}$$

$$F' = \{S \subseteq Q \mid S \cap F \neq \phi\}$$

# How do we compute $\varepsilon - closure(S)$?

$\varepsilon - closure(S):$

push all elements of $S$ onto a stack

result $:= S$

while stack not empty

pop $q$ off the stack

for each $u \in \delta(q, \varepsilon)$

if $u \notin$ result

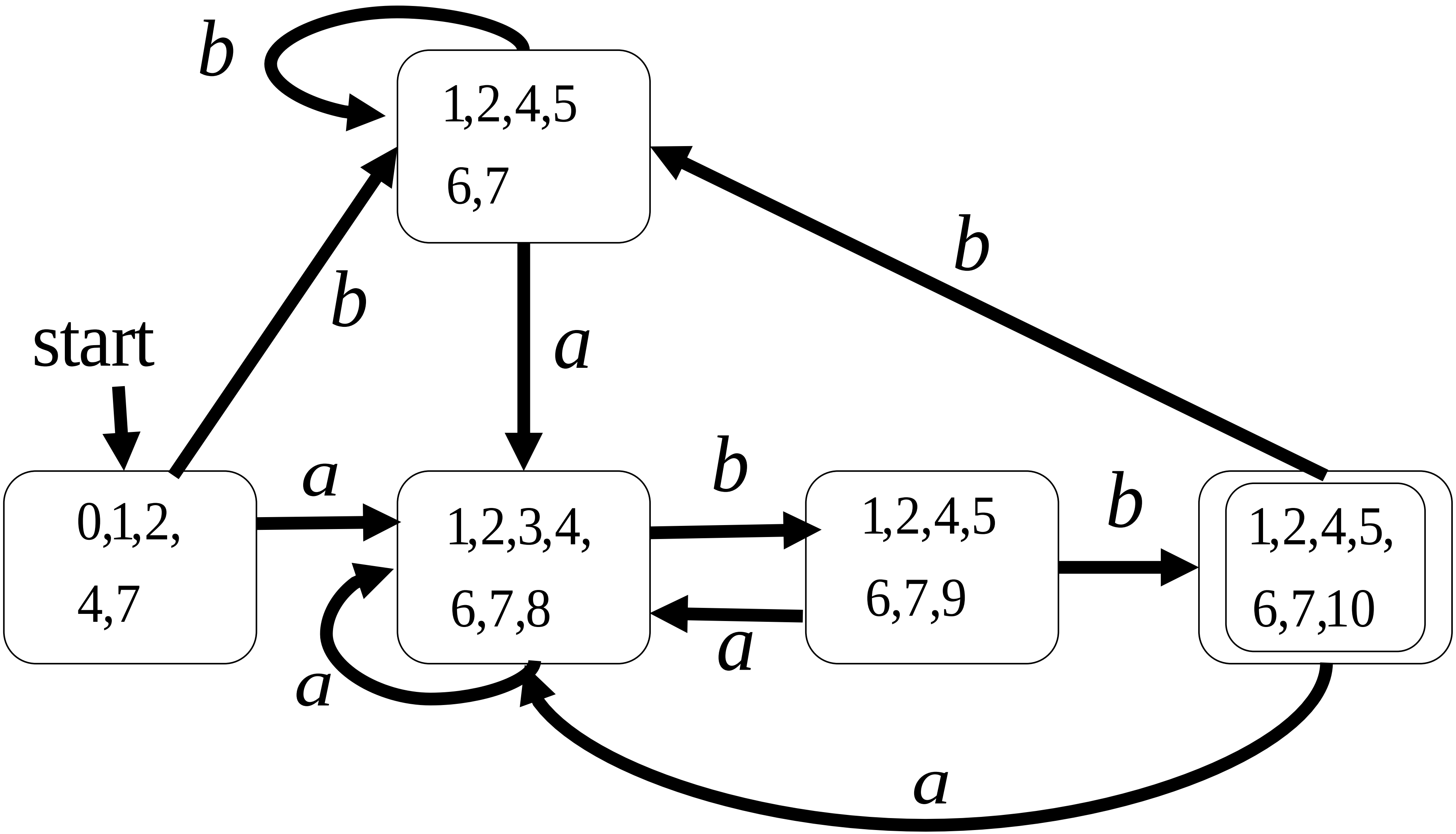
then result $:= \{u\} \cup$ result

push u on stack

return result

Look familiar?
It's just a version of transitive closure!

# $DFA(N((a+b)^{*}abb))$



# Traditional Regular Language Problem

$$\text{Given } e \text{ and } w, \text{ is } w \in L(e)?$$

**Solution : construct NFA from e, then DFA, then run the DFA on w.**

**But is this a solution to the “lexing problem?**

**No!**

# Something closer to the "lexing problem"

**Given** $e_1, e_2 \cdots, e_k$ **and** $w$

**find** $(i_1, w_1), (i_2, w_2), \ldots (i_n, w_n)$ **so that**

$$w = w_1 w_2 \ldots w_n$$ **and** $$\forall i \, \exists j \; w_i \in L(e_{i_j})$$

**and what else?**

The expressions are **ordered** by priority. Why? Is "if" a variable or a keyword? Need priority to resolve ambiguity (so "if" matched keyword RE before identifier RE.

We need to do a **longest** match. Why?
Is "ifif" a variable or two "if" keywords?

## … more formally

**Given** $e_1, e_2 \cdots, e_k$ **and** $w$

**find** $(i_1, w_1), (i_2, w_2), \ldots (i_n, w_n)$ **so that**

1) $w = w_1 w_2 \ldots w_n$

2) $\forall j, w_j \in L(e_{i_j})$

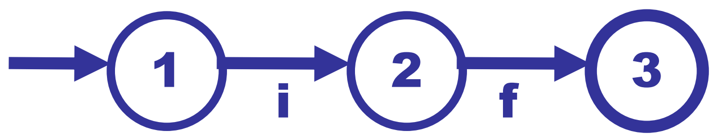
3) $w_j \in L(e_s) \rightarrow i_j \leq s$ (priority rule)

4) $\forall j : \forall u \in \text{prefix}(w_{j+1} w_{j+2} \cdots w_n), u \neq \varepsilon$

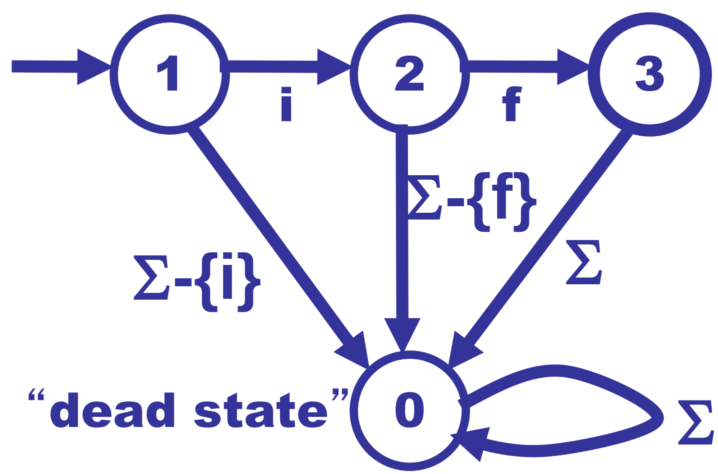
$\rightarrow \forall s, w_j u \notin L(e_s)$ (longest match)

# Define Tokens with Regular Expressions (Finite Automata)

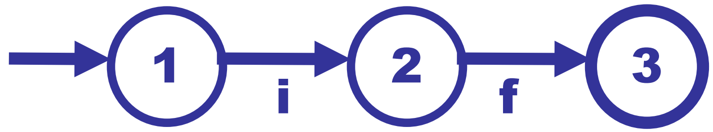
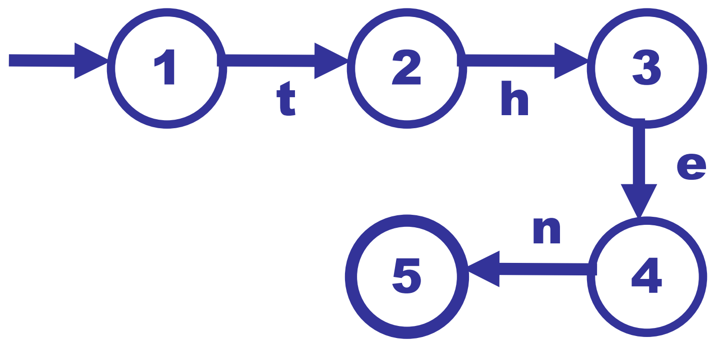
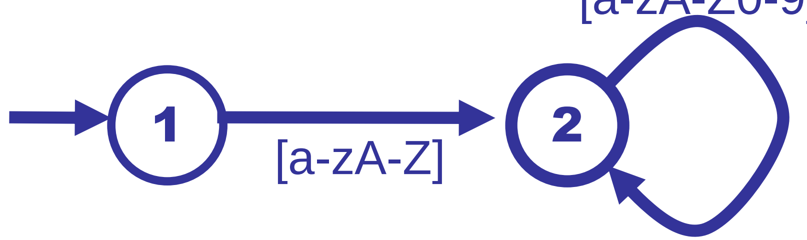
## Keyword: if



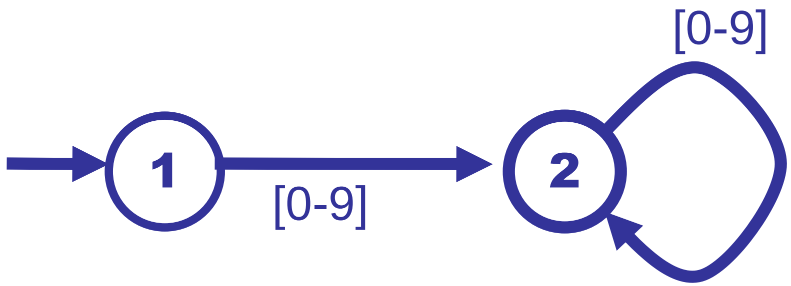
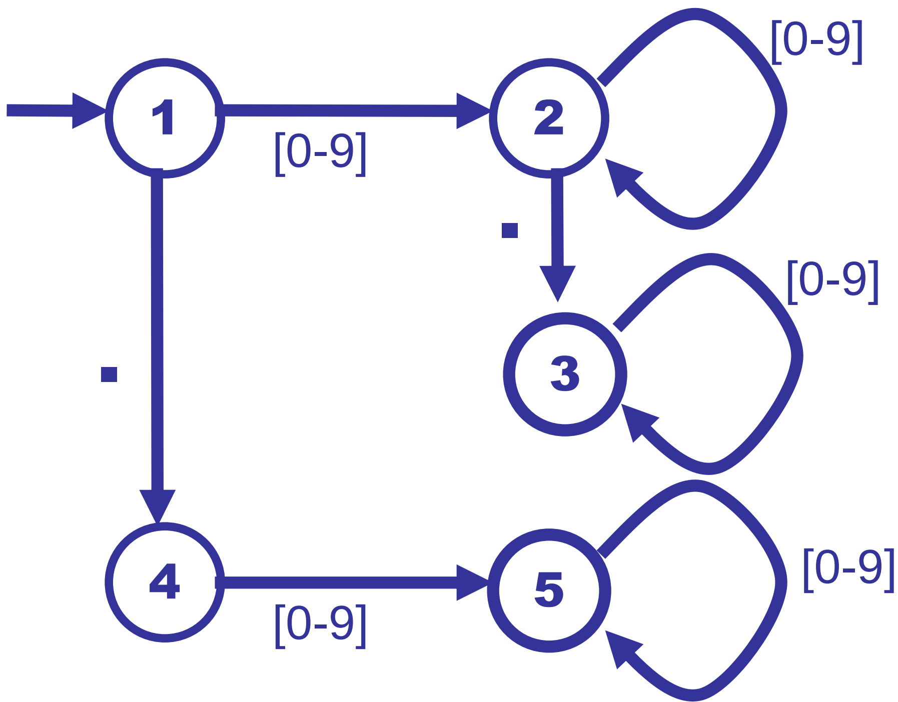

This FA is really shorthand for:



# Define Tokens with Regular Expressions (Finite Automata)

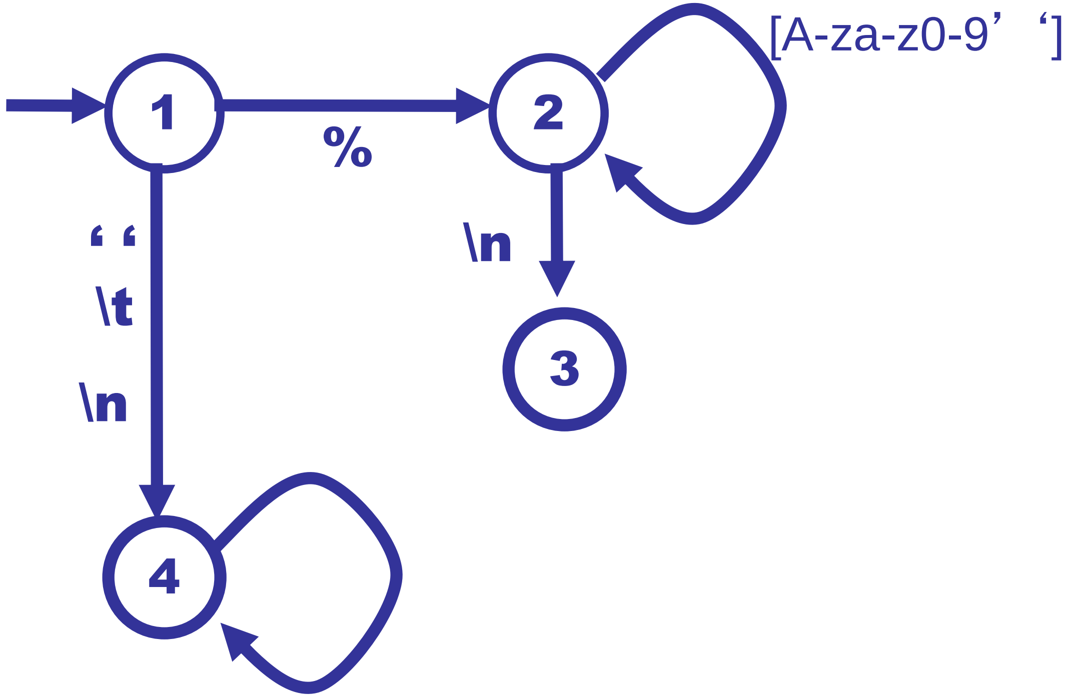
| Regular Expression | Finite Automata | Token |
|---|---|---|
| **Keyword: if** |   | **KEY(IF)** |
| **Keyword: then** |   | **KEY(then)** |
| **Identifier: [a-zA-Z][a-zA-Z0-9]*** |   | **ID(s)** |

# Define Tokens with Regular Expressions (Finite Automata)

| Regular Expression | Finite Automata | Token |
|---|---|---|
| **number:**<br>**[0-9][0-9]*** |  | **NUM(n)** |
| **real:**<br>**([0-9]+ ‘.’ [0-9]*)**<br>**\| ([0-9]* ‘.’ [0-9]+)** |  | **NUM(n)** |

# No Tokens for "White-Space"

## White-space with one line comments starting with %



# Constructing a Lexer

**INPUT:** $e_1, e_2 \cdots, e_k$

**an <u>ordered</u> list of regular expressions**
**Highest priority first, lowest last**

$$\Rightarrow \quad NFA_1, NFA_2 \cdots, NFA_k$$

use priority $\Rightarrow$ $NFA$

$\Rightarrow$ $DFA$

minimize $\Rightarrow$ $DFA$

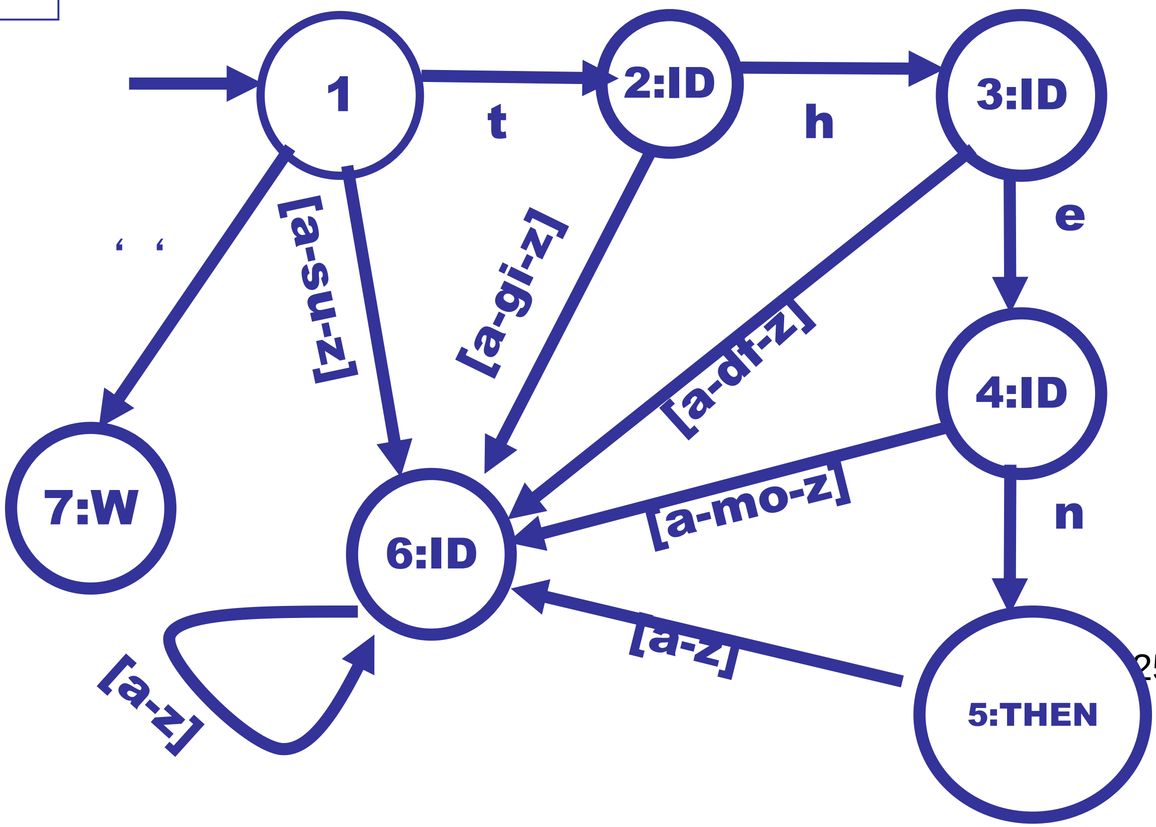
**(we have not discussed minimization)**

# Constructing a Lexer

(1) **Keyword : then**

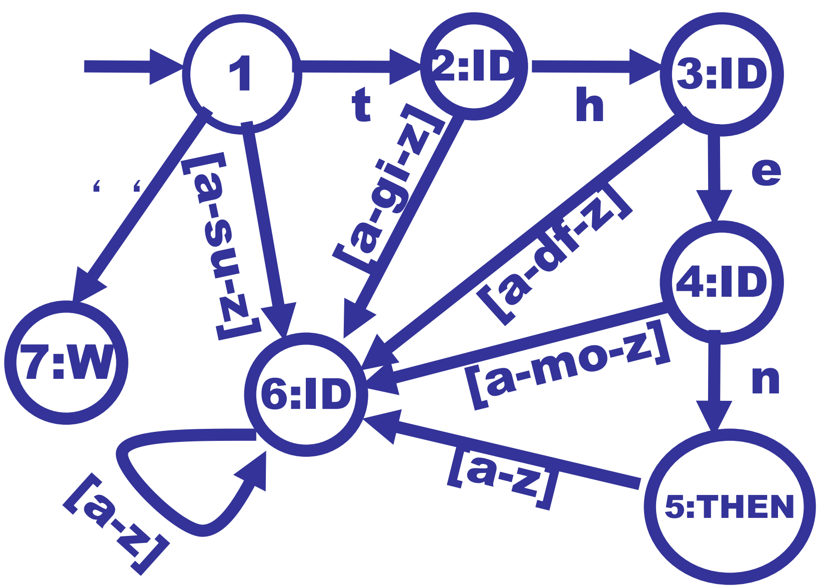
(2) **Ident : [a-z][a-z]***

(2) **White-space: ‘ ‘**



# What about longest match?

Start in initial state,
Repeat:
  (1) read input until dead state is reached. Emit token associated with last accepting state.
  (2) reset state to start state




| = current position, $ = EOF

| Input | current state | last accepting state | |
|---|---|---|---|
| \|then thenx$ | 1 | 0 | |
| t\|hen thenx$ | 2 | 2 | |
| th\|en thenx$ | 3 | 3 | |
| the\|n thenx$ | 4 | 4 | |
| then\| thenx$ | 5 | 5 | |
| then \|thenx$ | 0 | 5 | EMIT KEY(THEN) |
| then\| thenx$ | 1 | 0 | RESET |
| then \|thenx$ | 7 | 7 | |
| then t\|henx$ | 0 | 7 | EMIT WHITE(‘ ‘) |
| then \|thenx$ | 1 | 0 | RESET |
| then t\|henx$ | 2 | 2 | |
| then th\|enx$ | 3 | 3 | |
| then the\|nx$ | 4 | 4 | |
| then then\|x$ | 5 | 5 | |
| then thenx\|$ | 6 | 6 | |
| then thenx$\| | 0 | 6 | EMIT ID(thenx) |