CPSC 323 Compilers and Languages

Instructor: Susmitha Padda

Lexical Analysis

2.1 Basics

2.2 Deterministic Finite State Machine (DFSM)

2.3 Non-Deterministic FSM (NFSM)

2.4 Equivalence of DFSM and NFSM

2.5 Regular Expression (RE)

2.6 RE and FSM

2.7 Application of DFSM to LA

2.8 Minimization of FSM States

2.1 Basics

- Tasks of Lexer:
  - tokenizing source, i.e. breaking up the source code into meaningful units called Tokens.
  - removing (overpass) comments
  - case conversion, if needed
  - interpretation of compiler directives (ex. include, ifdef etc.)
  - dealing with *pragmas* (i.e., significant comments)
  - saving source locations (file, line, column) for error messages

Token vs. Lexeme

- *Token* is a generic type of a meaningful unit
- *Lexeme* is the actual instance
- Ex. the source code is `if (a > b) then`

Token	Lexeme
Keyword	if
Separator	(
Identifier	a
Operator	>
Identifier	b
Keyword	then

Writing a Lexer

- Each token can be represented using regular expressions
- Regular expressions are represented using a finite state machine (FSM):
  - Deterministic Finite State Machine (DFSM) or Deterministic Finite Automaton (DFA)
  - Nondeterministic Finite State Machine (NFSM) or Nondeterministic Finite State Automaton (NFA)
  - The set of all DFSMs = the set of all NFSMs

Two kinds of Finite State Machines

Deterministic (DFSM):

- No state has more than one outgoing edge with the same label. [All previous FSM were DFSM.]

Non-deterministic (NFSM):

- States *may* have more than one outgoing edge with same label.
- Edges may be labeled with ε (epsilon), the empty string. [Note that some books use the symbol **λ.**]
- The automaton can make an ε epsilon transition *without* consuming the current input character.

2.2 Deterministic FSM (DFSM)

Def: **DFSM= (Σ, Q, q0, F, N)**

where **Σ** = a finite set of input symbols

Q = a finite set of states

$\mathbf{q_0} \in$ Q is the starting state

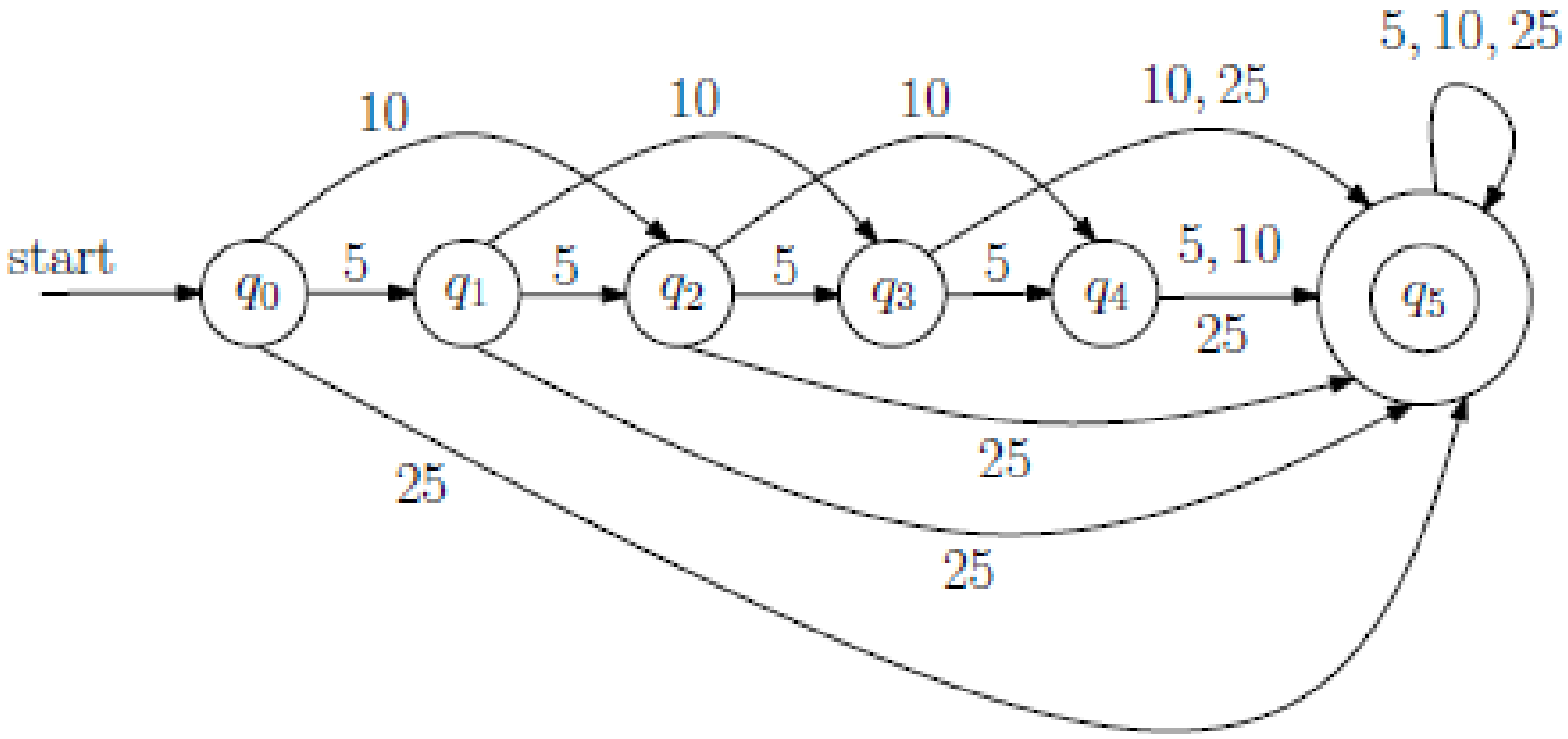
F ⊆ Q is a set of accepting (or final) states

N: (Q x Σ) -> Q is the State Transition Function (Given a state and an input -> goes to another state)

- **N** is a deterministic function, and for now, fully defined for each state and each symbol.
- In case **N** is not fully defined, we can consider an additional state called *null state* to complete the table definition of **N**.

An example: Controlling a toll gate

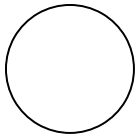
- Consider the problem of designing a “computer” that controls a toll gate
- When a car arrives at the toll gate, the gate is closed. The gate opens as soon as the driver has payed 25 cents. We assume that we have only three coin denominations: 5, 10, and 25 cents. We also assume that no excess change is returned.
- After having arrived at the toll gate, the driver inserts a sequence of coins into the machine. At any moment, the machine has to decide whether or not to open the gate, i.e., whether or not the driver has paid 25 cents (or more).
- In order to decide this, the machine is in one of the following six states, at any moment during the process:
  - The machine is in state q0, if it has not collected any money yet.
  - The machine is in state q1, if it has collected exactly 5 cents.
  - The machine is in state q2, if it has collected exactly 10 (=2x5) cents.
  - The machine is in state q3, if it has collected exactly 15(=3x5) cents.
  - The machine is in state q4, if it has collected exactly 20(=4x5) cents.
  - The machine is in state q5, if it has collected 25(=5*5) cents or more.
- Initially (when a car arrives at the toll gate), the machine is in state q0
- What is the sequence of states reached if the driver presents the sequence (10,5,5,10) of coins?



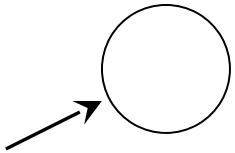
The machine (or computer) only has to remember which state it is in at any given time. Thus, it needs only a very small amount of memory: It has to be able to distinguish between any one of six possible cases and, therefore, it only needs a memory of $\lceil \log_2 6 \rceil = 3$ bits.

Finite State Machines viewed as Graphs

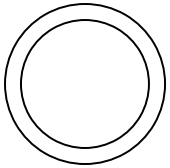
- A state



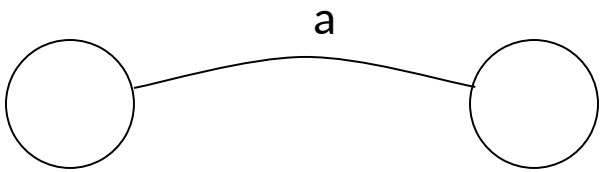
- The start state



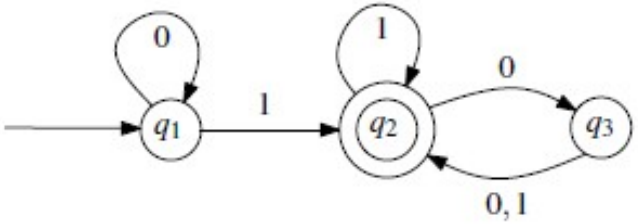
- An accepting state



- A transition



Example 1



- Starting state q1; accept state q2

- input string 00110: q1->q1->q2->q2->q3. State q3 is not an accept state so the machine rejects 00110
- Input string 1: q1->q2. State q2 is an accept state so the machine accepts 1
- Examples in textbook: 1101, 0101010
- What can you say about the strings accepted by the machine? To answer, one needs to look:
  - At the starting symbol
  - At the last symbol
  - If a certain substring needs to be there
  - If a certain number of the same symbol needs to be there
  - If only certain combinations (concatenated or not) are accepted

- What can you say about the DFA on the previous slide?
- The machine accepts every binary string having the property that there are an even number of 0s (including no 0) following the rightmost 1
- Empty string not accepted
- Strings consisting of 0s only are also rejected
- Strings with an odd number of 0s following the rightmost 1 are also rejected

DFSM for the toll gate example

Q = {q0, q1, q2, q3, q4, q5},

Σ = {5, 10, 25}

the start state is q0

F = {q5}

N is given by the following table:

	5	10	25
q0	q1	q2	q5
q1	q2	q3	q5
q2	q3	q4	q5
q3	q4	q5	q5
q4	q5	q5	q5
q5	q5	q5	q5

DFSM for Example 1

Q = {q1, q2, q3}
Σ = {0, 1}
the start state is q1
F = {q2}
N is given by the following table:

	0	1
q1	q1	q2
q2	q3	q3
q3	q2	q2

Example 2 of a DFSM

$Q = \{1,2,3,4\}$

$\Sigma = \{a, b, c\}$

$q_0 = 1$

$F = \{3,4\}$

N:

	a	b	c
-> $q_0 = 1$	2	1	3
2	4	2	1
3	3	4	2
4	1	3	2

- Graphical representation

Example 3: Candy Vending Machine

Vending Machine for a 25 cents candy. It accepts nickel(n), dime(d) and quarter(q) as input. It does not return change.

Q = {0,1,2,3,4,5,6}

Σ = {n,d,q}

$q_0 = 0$

F = {5,6}

N:

	n	d	q	
0	1	2	5	
1	2	3	6	
2	3	4	6	
3	4	5	6	
4	5	6	6	
5	6	6	6	
6	6	6	6	(return overflow)

Transitions

- Transition

$$s_1 \xrightarrow{a} s_2$$

- 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 (got stuck) => reject

How to Implement a FSM

A table-driven approach:

- Table:
  - one row for each state in the machine, and
  - one column for each possible character.
- Table[j][k]
  - which state to go to from state j on input character k,
  - an empty entry corresponds to the machine getting stuck.

Role of Graphical Representation

- Meaning? Representing the behavior of software in FSM allows the generation of software without writing actual program => Program Generator

Strings Accepted/Rejected by a DFSM

- A DFSM M *accepts* (recognizes) a string iff

1. the entire string has been read and
2. M is in any of the accept (final) states.

- Graphical representation of acceptance (also called *trace diagram*):

A string ω is accepted by M, iff there is a path from the starting state to any accept state whose edge spells ω.

Example 4

Given the following DFSM

M = (

Σ = {0,1}

Q = {A,B,C,D}

q_0 = A

F = {B,D}

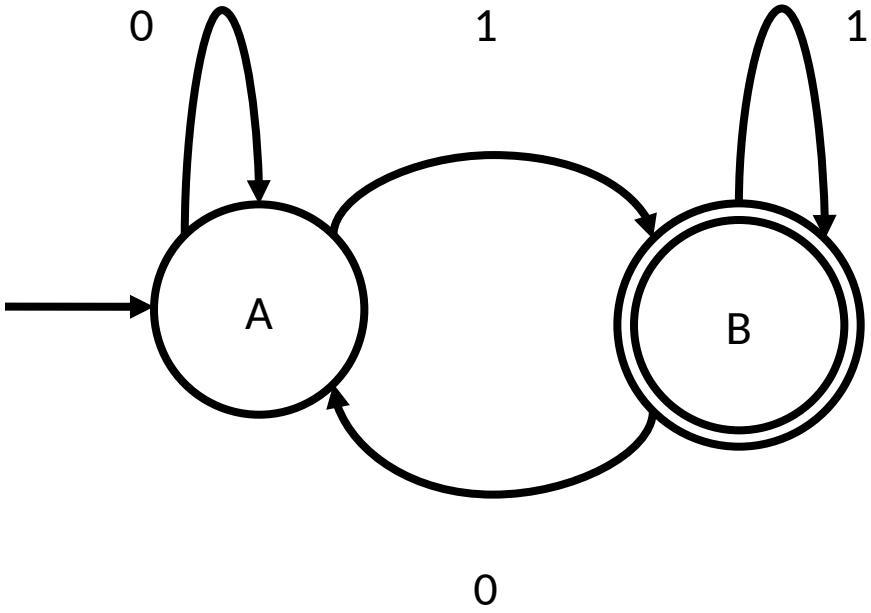
N:

	0	1
-> A	B	A
B	C	C
C	D	B
D	A	D

)

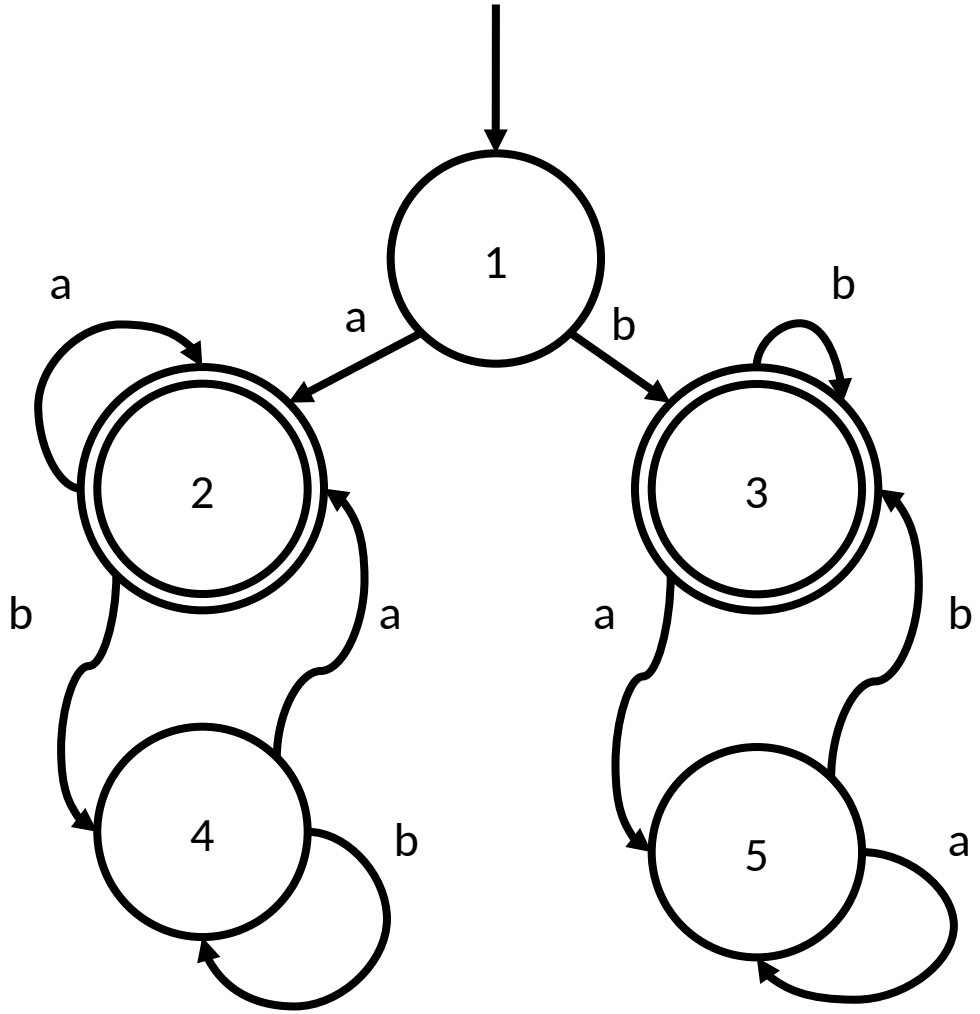
- Ex1. String 1 = 11001 accepted / not accepted?
- Ex2. String 2 = 011100 accepted / not accepted?

Example 5



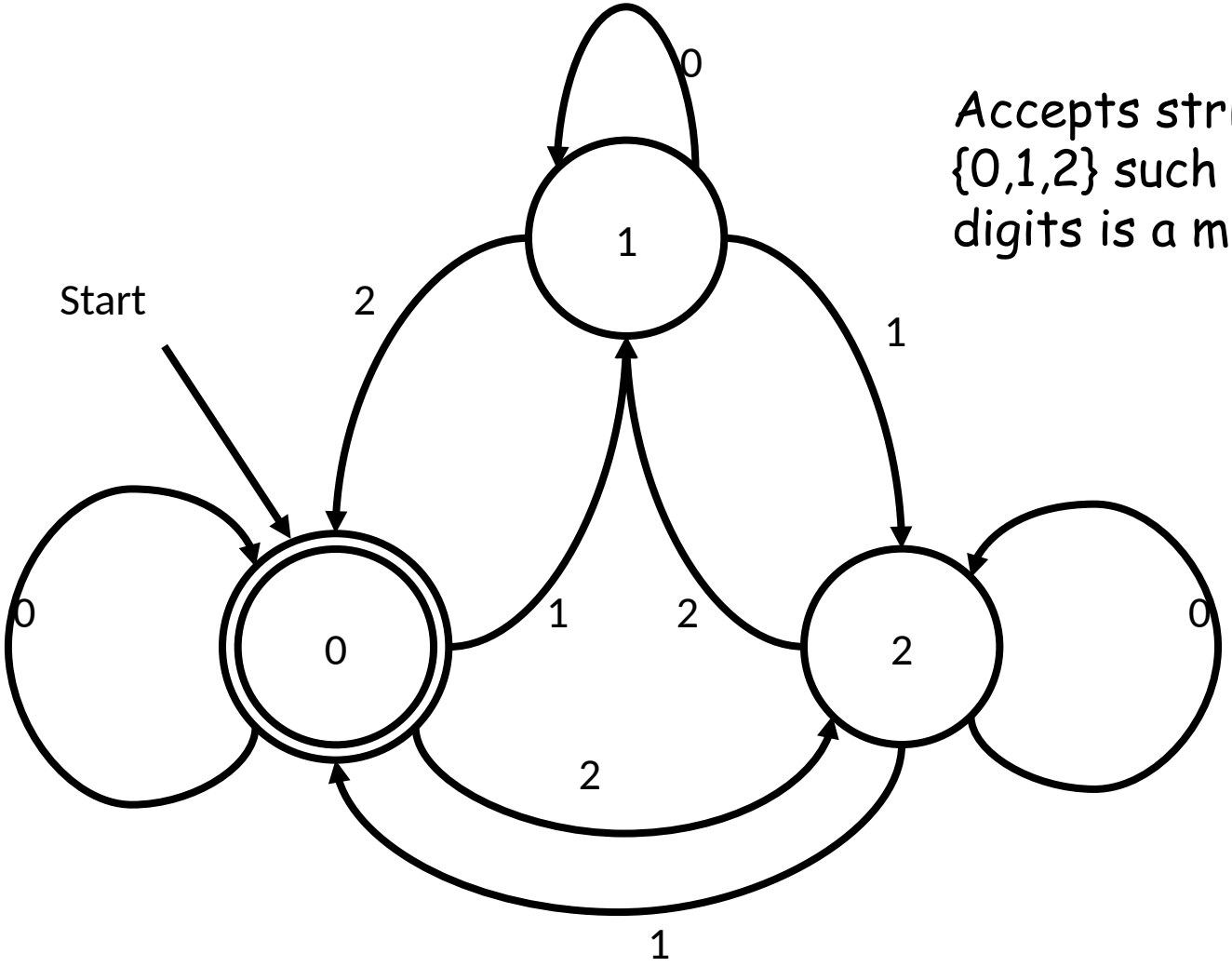
Accepts strings over alphabet {0,1} that end in 1

Example 6



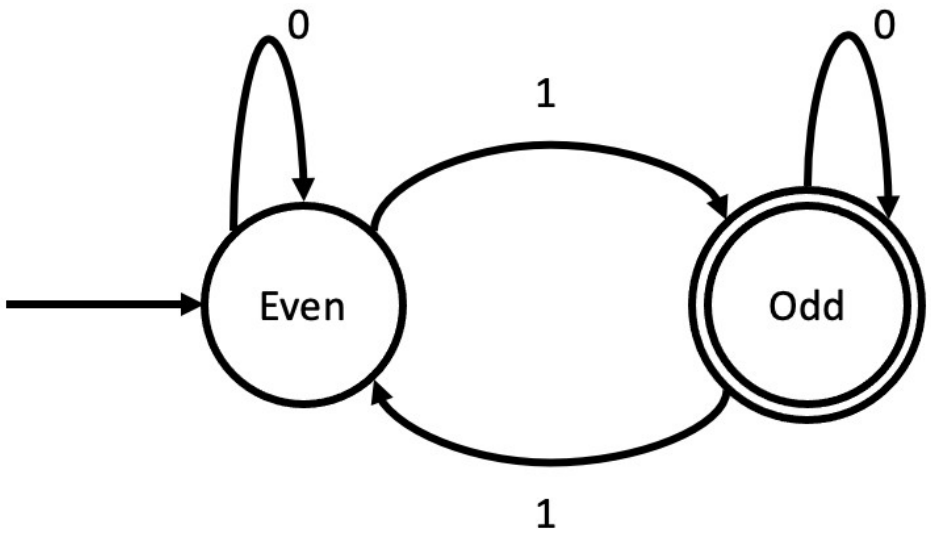
Accepts strings over alphabet {a,b} that begin and end with same symbol

Example 7



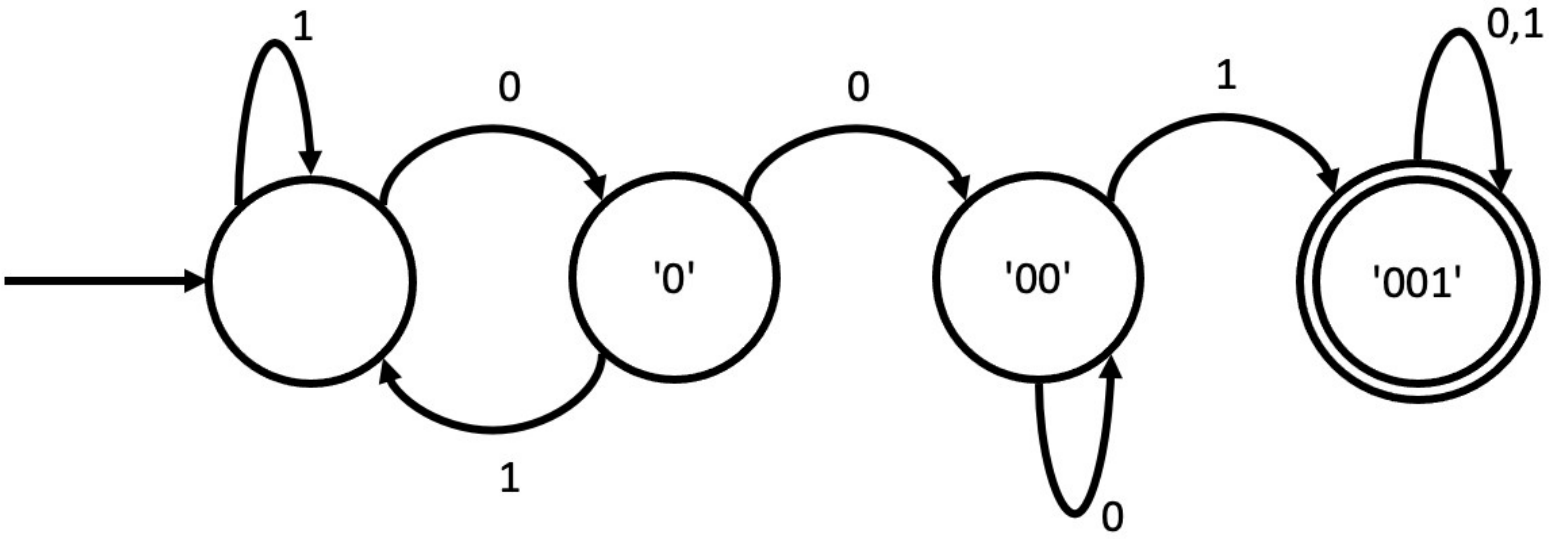
Accepts strings over {0,1,2} such that sum of digits is a multiple of 3

Example 8



Accepts strings over {0,1} that have an odd number of ones

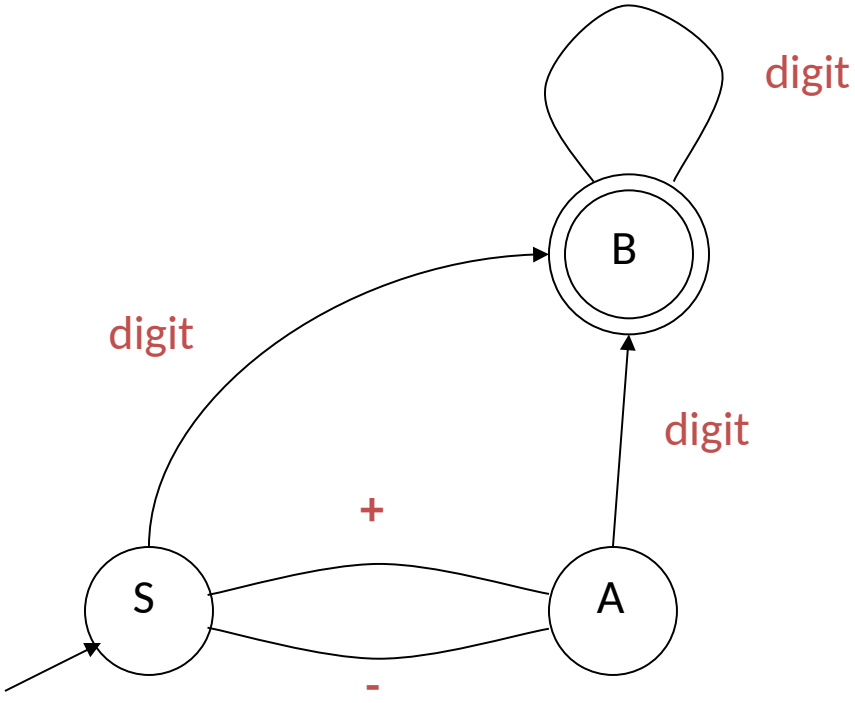
Example 9



Accepts strings over {0,1} that contain the substring 001

Example 10: Integer Literals

- FSM that accepts integer literals with an optional + or - sign:



Note regarding textbook notation

- If there is no mention of the starting state, then the first state (the first row) of the state transition table is the starting state
- The accepting state are either underlined or bolded

Implementing a DFSM

```
function DFSM (ω : string)
table = array [1..nstates, 1..nalphabets] of integer; /* Table N for the transitions*/
{
  state = 1; (the starting state)
  for i = to length (ω) do
    {
      col = char_to_col (ω[i]);
      state = table[state, col];
    }
  if state is in F then return 1  /* accept */
  else return 0
}
```

Note: the function char_to_col(ch) returns the column number of the ch in the table

ω= aca

Call DFSM (aca)

State = 1;

for i = 1 to 3 do =>

i = 1, col = 1, state = ntable (1,1) = 2

i = 2, col = 3, state = ntable (2,3) = 1

i = 3, col = 1, state = ntable (1,1) = 2

State 2 is not in F = {3,4}

=> return 0(false)

=> This string is NOT accepted

	a	b	c
q_0=1	2	1	3
2	4	2	1
3	3	4	2
4	**1**	**3**	**2**

Nondeterministic Finite State Machine (NFSM)

NFSM= (Σ, Q, q0, F, N)
where Σ = a finite set of input symbols

Q = a finite set of states

$q_0 \in Q$ is the starting state

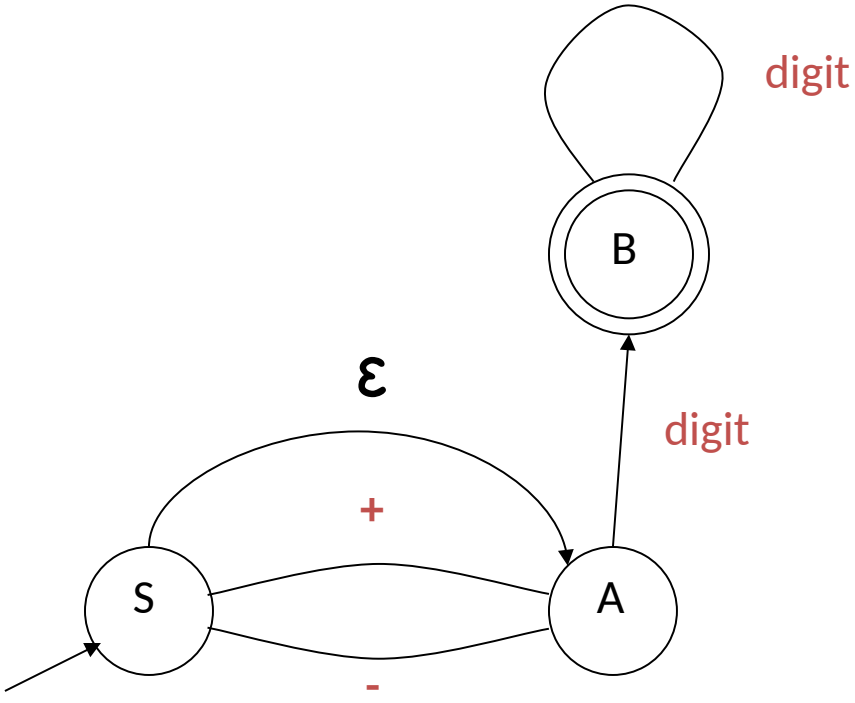
$F \subseteq Q$ is a set of accepting state

N: Q x (Σ U ε) -> P(Q)

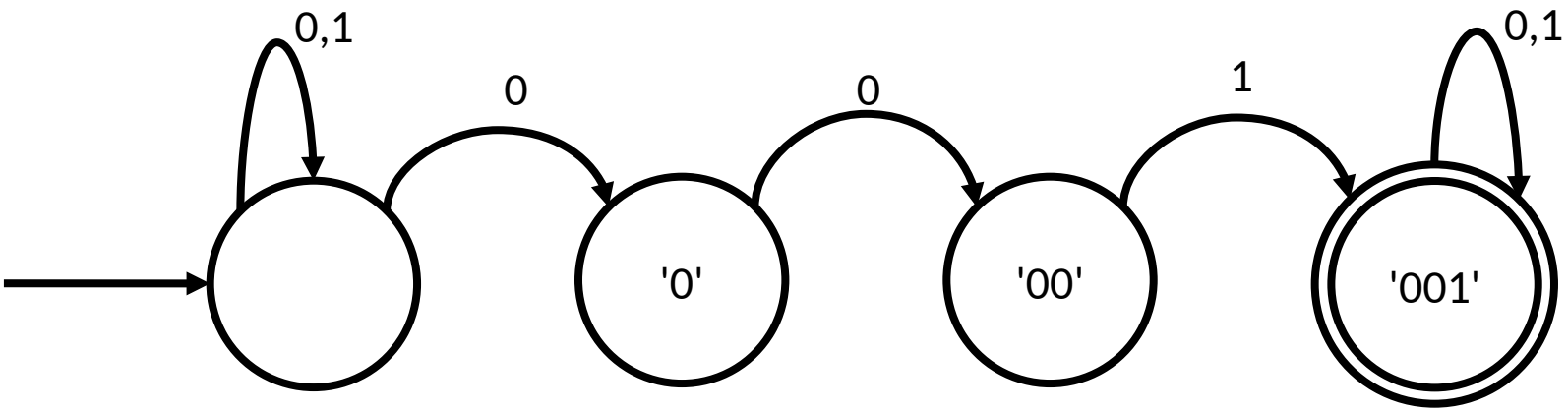
- The symbol ε (epsilon) means no input symbol.
- Input is expanded by epsilon => can go to a different state without reading any input (different from DFSM)
- Can go to multiple states given an input (different from DFSM)

Example 1 of NFSM

- integer-literal example:



Example 2 of NFSM



Accepts strings over {0,1} that contain the substring 001

String Accepted by a NFSM
(same as a DFSM)

- A NFSM M *accepts* (recognizes) a string iff

1. the entire string has been read and
2. M is in any of the accepting (final) states.

- Graphical representation of acceptance (also called *trace diagram*):

A string ω is accepted by M, iff there is a path from the starting state to any accepting state whose edge spells ω.

Example 3

- NFSM with Multiple States Transitions

M = (Σ = {a,b}, Q = {1,2,3}, q_0 = 1, F = {2,3}

N:

	a	b
->1	{1,2}	{1}
2	{2}	{1,3}
3	{2,3}	{1,3}

- Graphical Representation
- Is the string = baabab accepted / not accepted?

NSFM with Epsilon transitions

	a	b	ε
q_0= 1	{1,2}	{3}	{ }
2	{3}	{2,3}	{4}
3	{3,4}	{2}	{1,4}
4	{1}	{2}	{ }

- Graphical Representation
- Is the string = aba accepted / not accepted ?

Equivalence of DFSM and NFSM

- A *language* is any set of strings
- A *language over an alphabet* Σ is any set of strings made up only with characters from Σ
- A *language accepted by* M is the set of all strings over Σ that are accepted by M; we write it as L(M).
- Def: Two automata M_1 and M_2 are equivalent iff $L(M_1) = L(M_2)$, i.e iff they both accept the same language.
- Theorem: the class of languages accepted by DFSMs and NFSMs is the same. That is, for each NFSM there is an equivalent DFSM and vice versa.
- Every DFSM can be regarded as a NDFSM that just doesn't use the extra nondeterministic capabilities, so the class of languages accepted by DFSMs is included in the class of languages accepted by NFSMs
- Given any NFSM, an equivalent DFSM can be built (next slides) so the class of languages accepted by NFSMs is included in the class of languages accepted by DFSMs. Done.

Building a DFSM from a NFSM

- Claim: *For any NFSM M, we can construct an equivalent DFSM M' (No Proof in the class)*
- Given NFSM M = (Σ, Q, qo, F, N) convert in to a DFSM M' = ((Σ', Q', qo', F', N') as follows:
- Step 1: get rid of the epsilon transitions
- Step 2: solve the multiple states in an entry

Step 2: Solve the Multiple States in an Entry

$\Sigma' = \Sigma$

Q’ = the powerset of states in M including the empty set or null set [ ]

F’ = all states in Q’ that contains at least one accepting state in M

$q_{0'} = [q_0]$

N’ : such that

N’([P1, P2, …Pn], x $\in \Sigma'$) = N(P1, x) U N(P2,x) U…..U N(Pn,x) in [ ]

Example 1

Ex: M = (Σ={a,b), Q ={1,2,3}, q_0= 1, F={2},

N:	a	b
1	{1,2}	{1}
2	{2}	{1,3}
3	{2,3}	{1,3}

)

What is M'?

M’ = (Σ’={a,b),

Q’ ={[1],[2],[3],[1,2][1,3][2,3],[1,2,3],[]},

q0’= [1],

F’={[2],[1,2],[2,3],[1,2,3]},

N’:

	a	b
[1]		
[2]		
[3]		
[1,2]		?
[1,3]		
[2,3]		
[1,2,3]		
[ ]		

N’:

	a	b
[1]	[1,2]	[1]
[2]	[2]	[1,3]
[3]	[2,3]	[1,3]
[1,2]	[1,2]	[1,3]
[1,3]	[1,2,3]	[1,3]
[2,3]	[2,3]	[1,3]
[1,2,3]	[1,2,3]	[1,3]
[ ]	[ ]	[ ]

- If Q has n states, then the power set $Q' = 2^Q$ has 2^n states
- Example: n = 5 then Q' has 32 states
- Impractical
- Some states are unnecessary ex. [ ], [2] => can never be reached from starting state
- Remove the states that cannot be reached from the starting state
- Build the DFSM with the minimum number of states. Not presented in class.

Step 1: Get Rid of Epsilon Transitions

- ε-closure of a state q ∈ Q, **ε(q),** is the set of all states that can be reached from q by means of epsilon transitions including q, i.e. traveling along zero or more **ε** arrows
- ε-closure of a subset of states R ⊆ Q, **ε(R)** = **{q** that can be reached from some r **∈ R** by traveling along zero or more **ε** arrows **}**

Input: a NFSM N = $(Q, \Sigma, \delta, q_0, F)$

Output: a DFSM M = $(Q', \Sigma, \delta', q_0', F')$

$Q' = 2^Q$

$\delta' : Q' \times \Sigma \rightarrow Q'$

$q_0' = \varepsilon(q_0)$

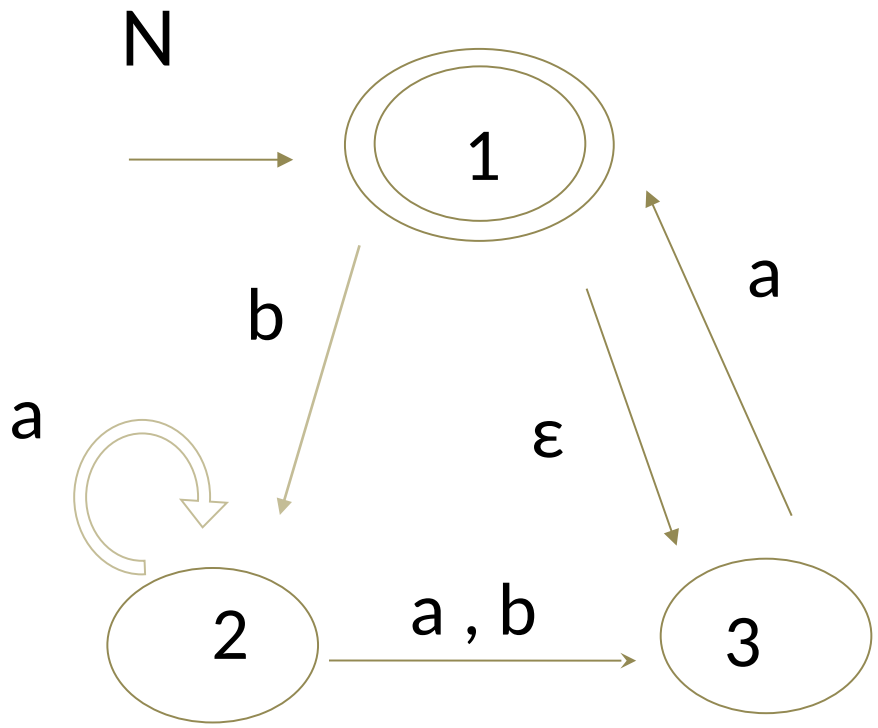
$F' = \{ R \in Q' \mid f \in R \text{ for some } f \in F \}$

$$\delta'(R,\sigma) = \bigcup_{r \in R} \varepsilon(\delta(r,\sigma))$$

- Remove states that cannot be reached from the starting state of the DFSM

Example 2

Given an NFSM N = ({1,2,3}, {a.b}, δ , {1}, {1}), construct an equivalent DFSM M

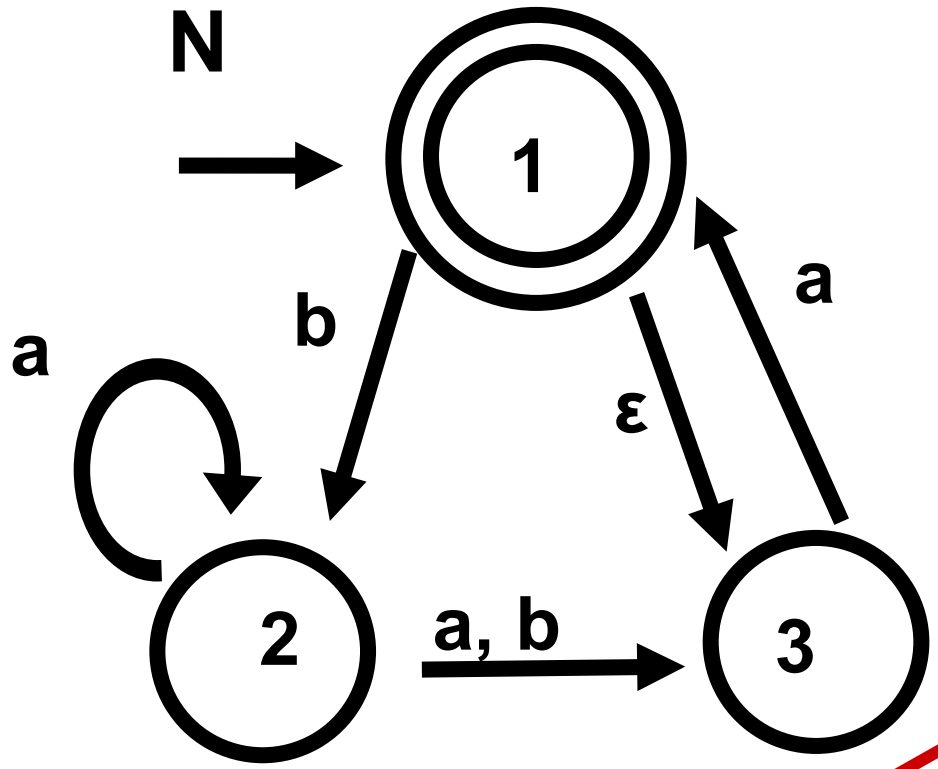


ε({1}) = {1,3}

$N = (Q, \Sigma, \delta, Q_0, F)$

Given: NFA $N = (\{1,2,3\}, \{a,b\}, \delta, \{1\}, \{1\})$

Construct: equivalent DFA $M = (Q', \Sigma, \delta', q_0', F')$



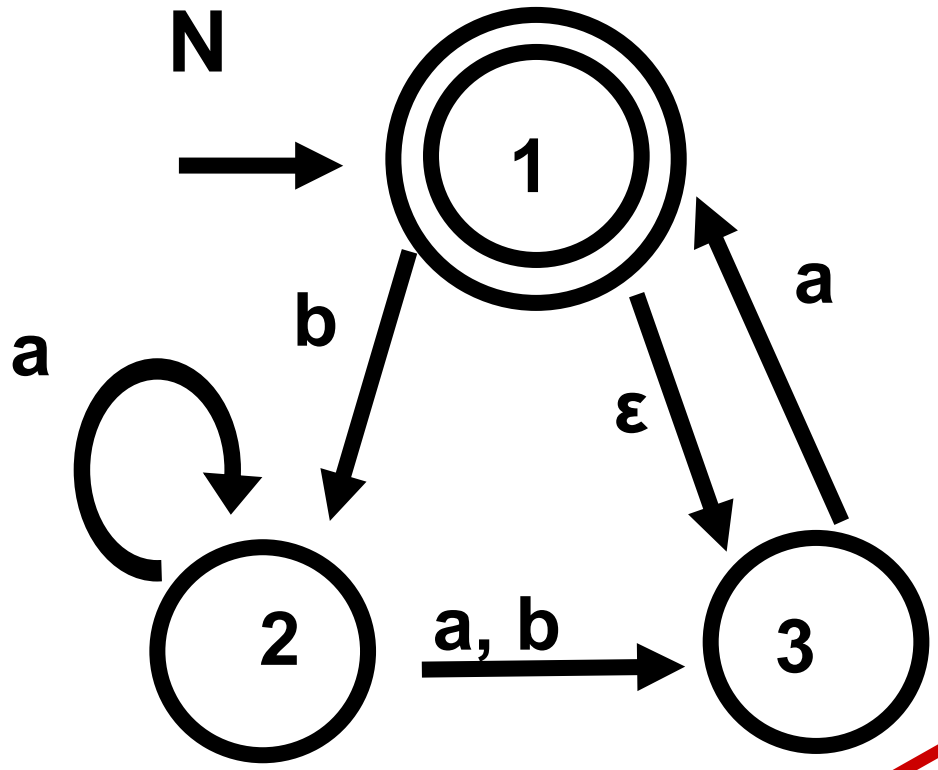
$q_0' = \varepsilon(\{1\}) = \{1,3\}$

δ'	a	b
∅	∅	∅
{1}	∅	{2}
{2}	{2,3}	{3}
{3}	{1,3}	∅
{1,2}	{2,3}	{2,3}
{1,3}	{1,3}	{2}
{2,3}	{1,2,3}	{3}
{1,2,3}	{1,2,3}	{2,3}

$N = (Q, \Sigma, \delta, Q_0, F)$

Given: NFSM $N = (\{1,2,3\}, \{a,b\}, \delta, \{1\}, \{1\})$

Construct: equivalent DFSM $M = (Q', \Sigma, \delta', q_0', F')$



$q_0' = \varepsilon(\{1\}) = \{1,3\}$

δ'	a	b
∅	∅	∅
{1}	∅	{2}
{2}	{2,3}	{3}
{3}	{1,3}	∅
{1,2}	{2,3}	{2,3}
{1,3}	{1,3}	{2}
{2,3}	{1,2,3}	{3}
{1,2,3}	{1,2,3}	{2,3}

Example 2

Given: NFSM N = ({1,2,3}, {a,b, c}, δ , {1}, {3})

δ	a	b	c	ε
->1	Ø	{2}	{3}	{2,3}
2	{1}	{3}	{1, 2}	Ø
<u>3</u>	Ø	Ø	Ø	Ø

Construct: equivalent DFSM M= (Q′, Σ, δ′, q_0′, F′)

2.5 Regular expressions

- Are a means to describe languages.
- The class of languages that can be described by regular expressions coincides with the class of regular languages.
- REs are a more compact way to define a language that can be accepted by an FA
- Example: the expression (0 U 1)01*
  - The language described by this expression is the set of all binary strings
    1. that start with either 0 or 1 (this is indicated by (0 U 1)),
    2. for which the second symbol is 0 (this is indicated by 0), and
    3. that end with zero or more 1s (this is indicated by 1).
  - That is, the language described by this expression is {00, 001, 0011, 00111, . . . , 10, 101, 1011, 10111, . . .}.

contd.

Def: Let Σ be a non-empty alphabet.

1. ε is a regular expression.
2. ∅ is a regular expression.
3. For each symbol in the alphabet $a \in \Sigma$, a is a regular expression.
4. If R1 and R2 are regular expressions, then their union R1 U R2 is a regular expression; instead of U one can use |
5. If R1 and R2 are regular expressions, then their concatenation R1R2 is a regular expression.
6. If R is a regular expression, then Kleene star (concatenation of zero or more strings from R) R* is a regular expression.

Another notation used is $R^+ = R\,R^* = R^*R$

Example 1

- Given Σ = {0,1}, examples of REs are:

0 is a RE

1 is a RE

01 is a RE

10 is a RE

0U1 is also RE,

1^* is RE (0 or more repetitions of 1)

1^+ is RE (1 or more repetitions of 1)

0^*1

$(0U1)^*$

$(0U1)^*101(0U1)^*$

- Draw the NFA in each case

REs

- Let Σ be a non-empty alphabet.

1. The regular expression ε describes the language $\{\varepsilon\}$.
2. The regular expression $\varnothing$ describes the language $\varnothing$.
3. For each $a \in \Sigma$, the regular expression a describes the language {a}.
4. Let R_1 and R_2 be regular expressions and let L_1 and L_2 be the languages described by them, respectively. The regular expression $R_1 \cup R_2$ describes the language $L_1 \cup L_2$.
5. Let R_1 and R_2 be regular expressions and let L_1 and L_2 be the languages described by them, respectively. The regular expression R_1R_2 describes the language L_1 concatenated with L_2 , $L_1 L_2$
6. Let R be a regular expression and let L be the language described by it. The regular expression R* describes the language L*

Examples

- The regular expression (0U ε)(1U ε) describes the language {01, 0, 1, ε}.
- The regular expression 0 U ε describes the language {0, ε }
- The regular expression 1^* describes the language {ε, 1, 11, 111, . . .}.
- The regular expression (0 U ε)1^* describes the language {0, 01, 011, 0111, . . . , , 1, 11, 111, . . .}. This language is also described by the regular expression 01^* U 1^*. So we write (0 U ε)1^* = 01^* U 1^*
- The regular expression 1^* ∅ describes the empty language, i.e., the language ∅
- The regular expression $∅^*$ describes the language {ε}.
- Read Th. 2.7.4 and remember the rules 1-16

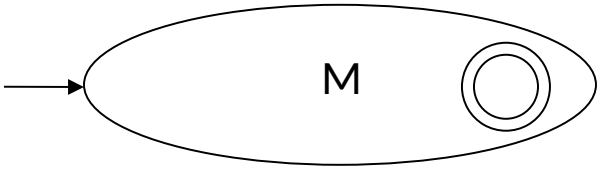
2.6 FSMs and REs

- Our goal: develop a procedure for generating state tables for a lexical analyzer
- A finite state machine is a good “visual” aid but it is not very suitable as a specification (its textual description is too clumsy)
- Regular expressions are a suitable specification; a more compact way to define a language that can be accepted by an FSM
- Are used to give the lexical description of a programming language
  - define each “token” (keywords, identifiers, literals, operators, punctuation, etc.)
  - define white-space, comments, etc.
    - these are not tokens, but must be recognized and ignored

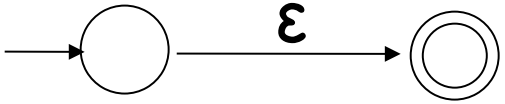
Equivalence between FSMs and REs

- Theorem: There exists a FSM for each RE. No proof in class) and vice versa.
- Q: How can we construct a FSM for a given RE?
- A: Using Thompson Construction Method
  - Idea: Build an FSM from smaller FSMs inductively => Provides a way for each RE rule
  - For each kind of RE, define a NFSM
  - Edges not marked are ε
  - Next convert each resulting NFSM to a DFSM

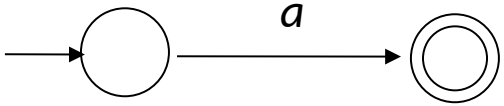
- For each kind of regular expression, define a NFSM
  - Notation: NFSM for regular expression M



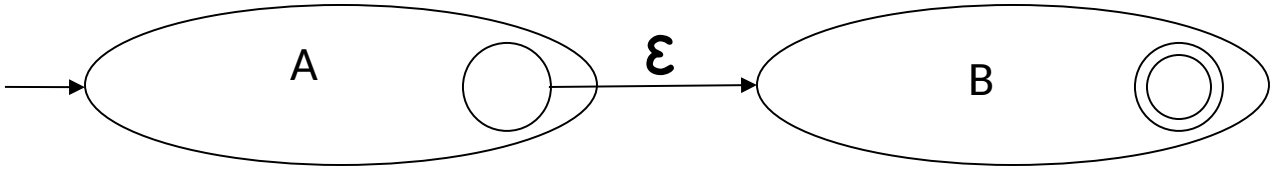
- For ε



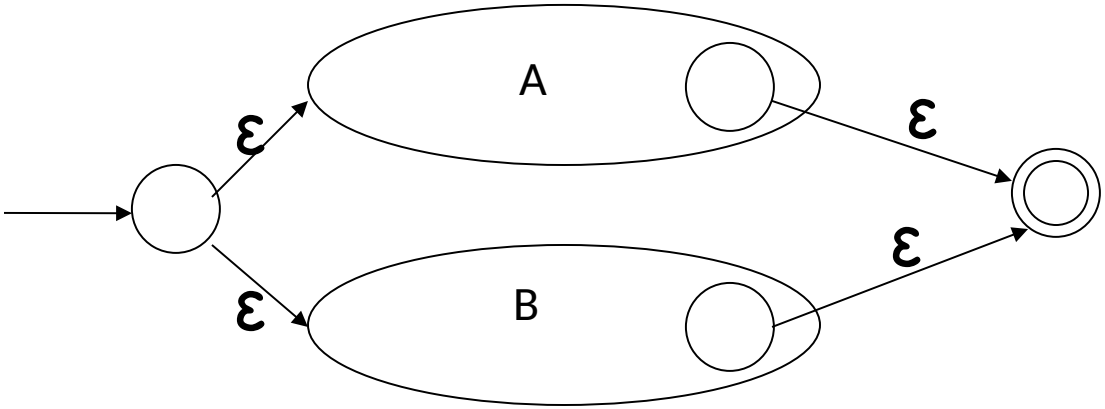
- For input *a*



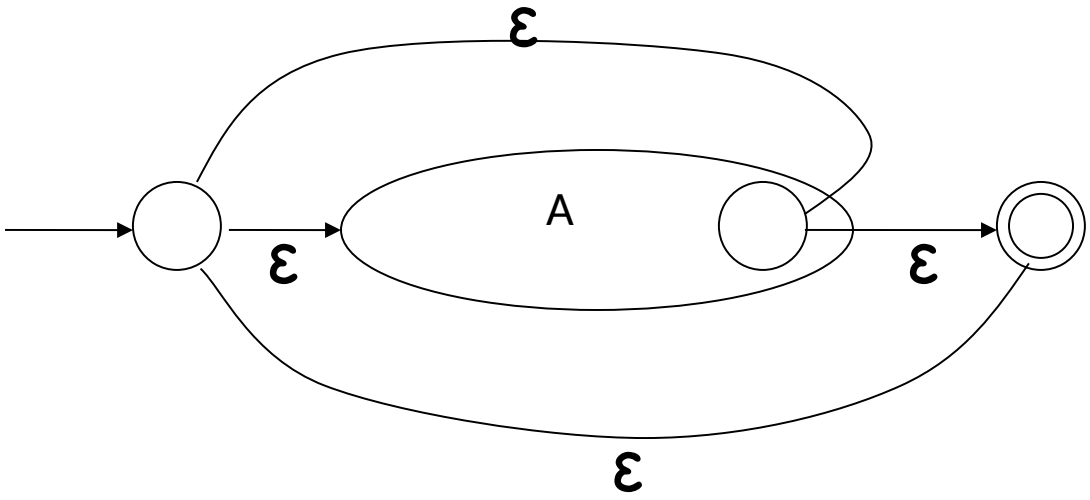
- For A B



- For A U B

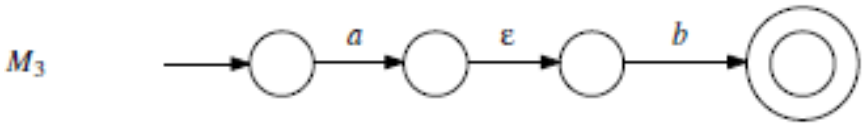


- For A*

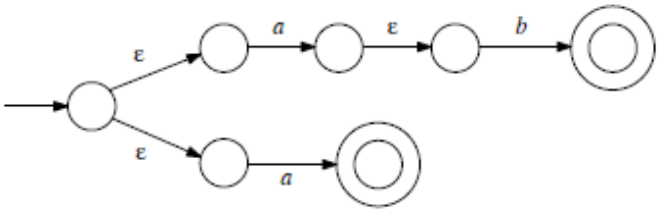


Example 1

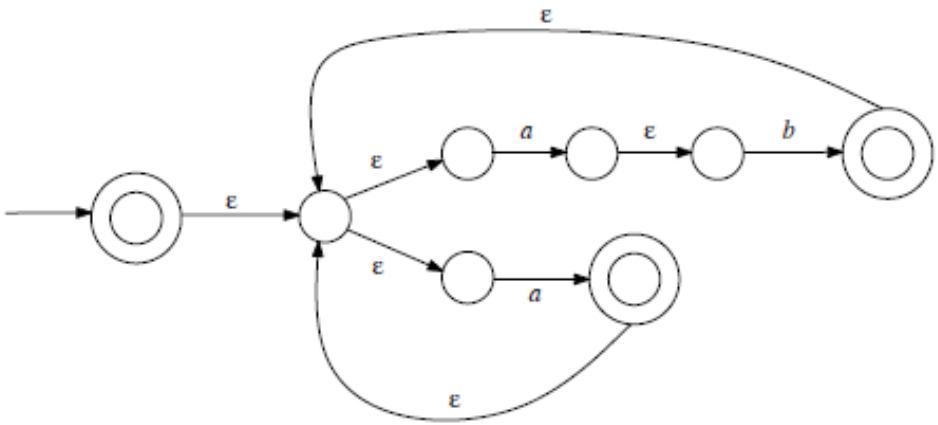
- RE (ab U a)*, construct the NFSM
- The NFSM for ab:



- The NFSM for ab U a:



- The final NFSM

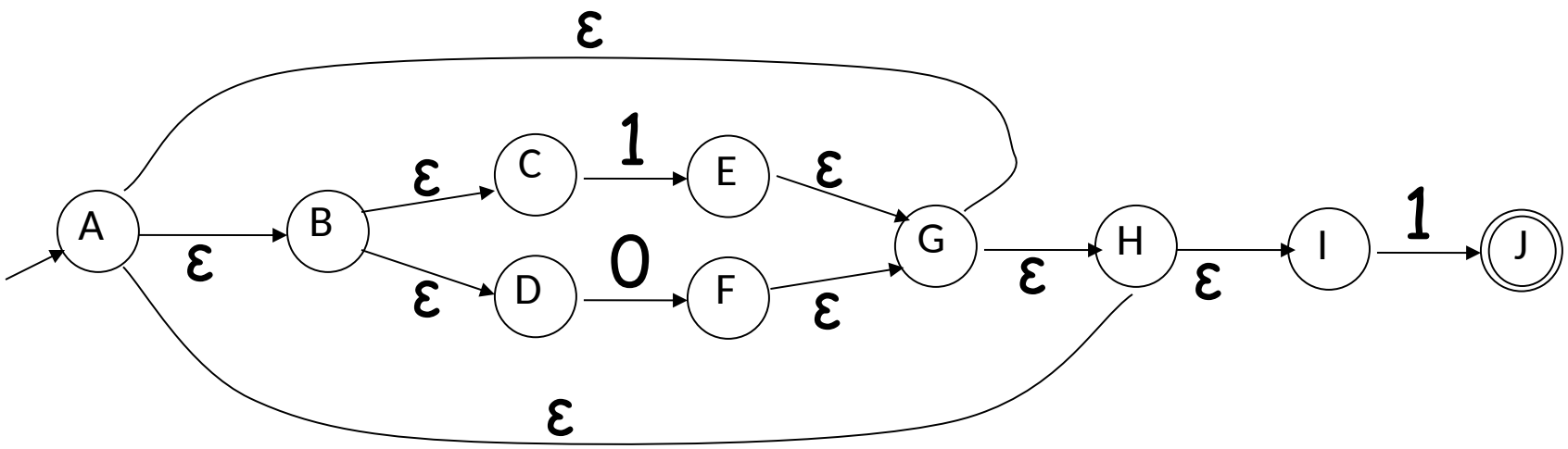


Example 2

- Consider the regular expression

$$(1|0)^*1$$

- The NFSM is



Example 3

- Build a FSM for RE (ab | ab*a)+

Examples

- (ε | + | -) (0|d*) . (0 | d)* -> legal floating point number

- (_ | letter) (letter | digit | 0 | _) * -> an identifier (other definitions of id)

Operands of a regular expression

- Operands are same as labels on the edges of an FSM
  - single characters, or
  - the special character ε (the empty string)

- "letter" is a shorthand for
  - *a* | *b* | *c* | … | *z* | *A* | *B* | *C* | … | *Z*
- "digit“ is a shorthand for
  - *0* | *1* | *2* | … | *9*
- sometimes we put the characters in quotes
  - necessary when denoting | * ( )

Precedence of | . * operators.

Regular Expression Operator	Analogous Arithmetic Operator	Precedence
|	plus	lowest
	times	middle
*	exponentiation	highest

- Consider regular expressions:
  - letter letter | digit*
  - letter (letter | digit)*

TEST YOURSELF

Question 1: Describe (in English) the language defined by each of the following regular expressions:

letter (letter* | digit*)

(letter | _) (letter | digit | _)*

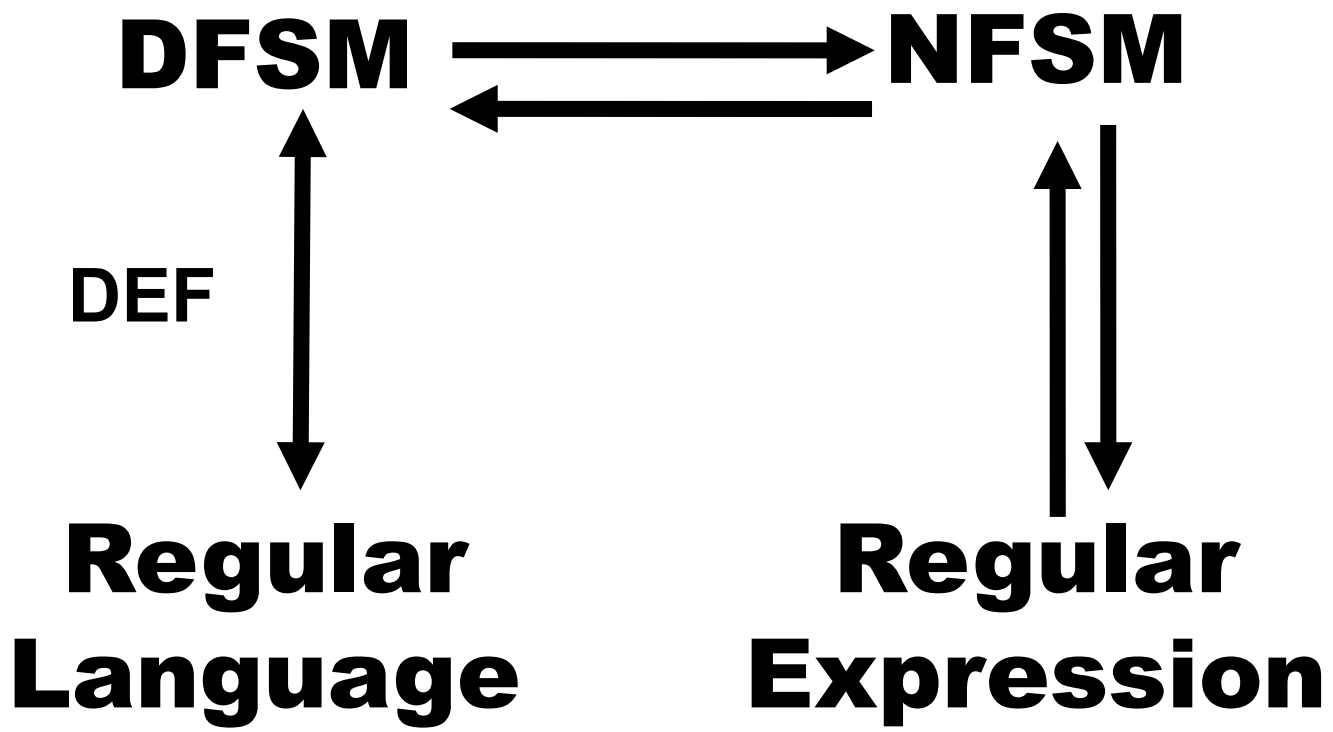
digit* "." digit*

digit digit* "." digit digit*

TEST YOURSELF

Question 2: Write a regular expression for each of these languages:

- The set of all C++ reserved words
  - Examples: if, while, for, class, int, case, char, true, false
- C++ string literals that begin with ” and end with ” and don’t contain any other ” except possibly in the escape sequence \”
  - Example: ”The escape sequence \” occurs in this string”
- C++ comments that begin with /* and end with */ and don’t contain any other */ within the string
  - Example: /* This is a comment * still the same comment */


DFSM
NFSM
DEF
Regular Language
Regular Expression

Complement of a language accepted by a DFA

- To build a DFA that accepts the complement of a language accepted by a DFA:
  - You need to draw the DFA that accepts the language
  - Change all final states into non-final states and all non-final states into final states.
- Example: if you have q0, q1, q2 and q3 as states of the DFA, q0 the starting state and q2 as the only final state, then without changing any labels of the arcs, make q0, q1 and q3 as final states, and q2 is non-final state. But you must make sure that you have a DFA, this will not work correctly for an NFA.

Describing Tokens as Res or FSMs

- Tokens are described using REs
- Tokens can be described using FSMs with some modifications:
  - white space is to be ignored unless it marks the end of a token
  - An accepting state must announce a token so an acceptance state can be reached only when the token has been read
  - An accepting state must tell us what token was found so we would like to have one accepting state for each token
  - Some character strings are identifiers, while others are keywords

Problems

- Some tokens can be prefixes of other tokens
- Some HLL use multiple-character delimiters for comments /* … */ (in C)
- Character-string constants are delimited by single quotes but single quotes can be found inside

2.7 Application to Lexical Analysis

Steps for constructing LA:

Step 1) Write tokens in terms of REs

Step 2) Build a NFSM that recognizes REs

Step 3) Convert the NFSM to a DFSM

Step 4) Modify the DFSM to recognize individual token

- One accepting state per token type
- Meaning of acceptance changes (Not the entire string – source code)
- Sometimes a char pointer needs to back up
- Ex. bcd*xyz need to back up by un character (Unget in C or C++)

Build a LA for identifier and Integer token

- An identifier has a letter followed by any number of letters or digits.
- An integer has any number of digits. (Rem. 0000 is allowed)

Step1) RE for Identifier is l(l | d) * and RE for an integer is d+

Step2) Build a NFSM for identifier and integer

Step 3) Convert the NFSM into a DFSM

Step 4) Change the DFSM to recognize each token

- Make sure there is one accepting state per token type