CPSC 323: Compilers and Languages

Chapter 2 – Lexical Analysis

Mr. Param Venkat Vivek Kesireddy

Teaching Associate

Mail: pkesireddy@fullerton.edu

nputs from Prof Doina Bein & Prof James Choi

Lexical Analysis

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 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
Symbol	(
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)

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

q_0 ∈ Q is the starting state

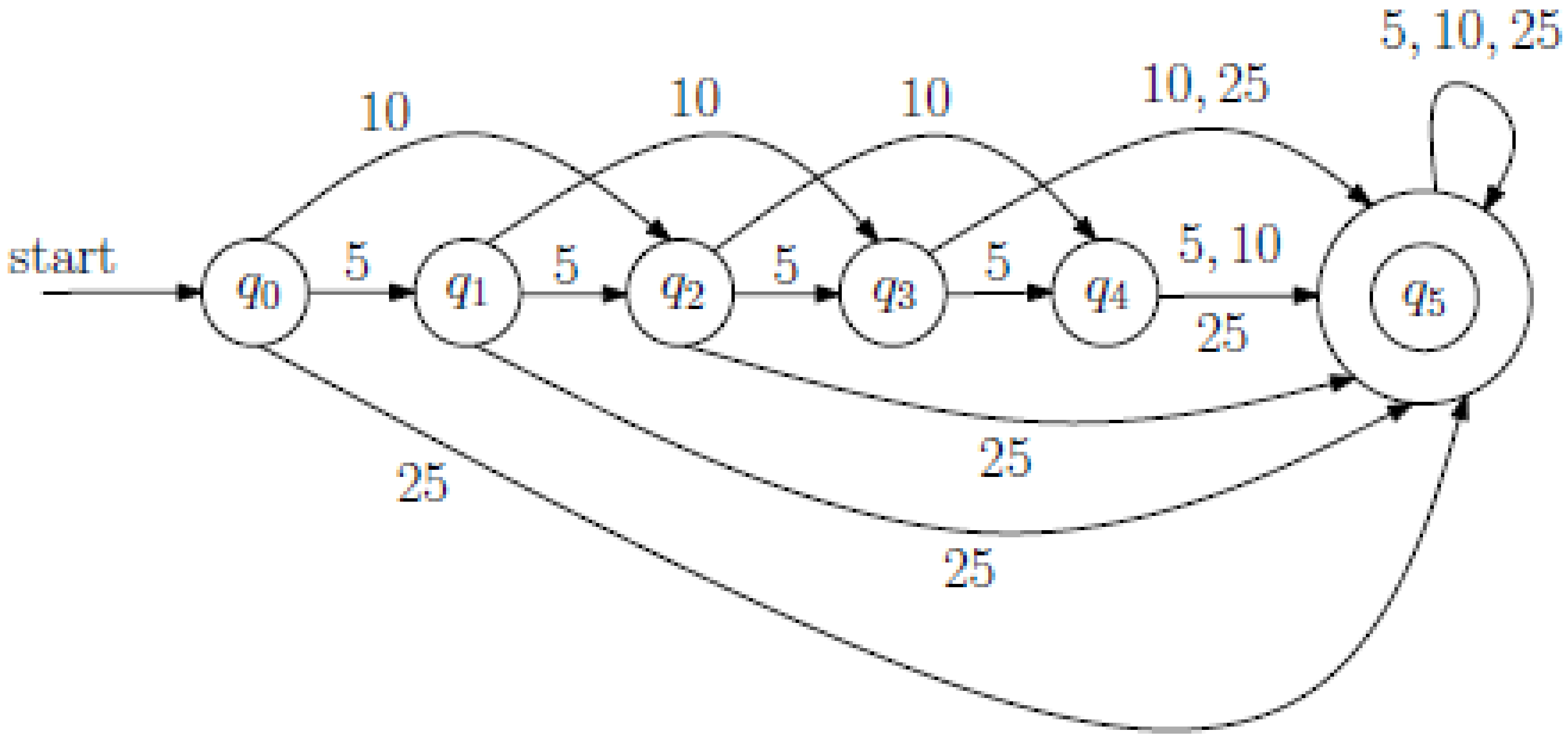
F ⊆ Q is a set of accepting (or final) states

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**.

Example 1 - Controlling a toll gate

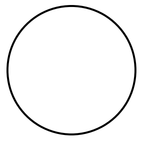
- Consider the problem of designing a “computer” that controls a toll gate
- When a car arrives at the toll gate, the gate is closed. The gate opens as soon as the driver has paid 25 cents. We assume that we have only three-coin denominations: 5, 10, and 25 cents. We also assume that no excess change 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?



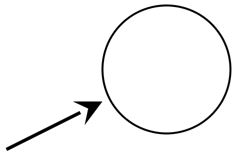
The machine (or computer) only has to remember which state it is in at any given time. Thus, it needs only a very small amount of memory: It has to be able to distinguish between any one of six possible cases and, therefore, it only needs a memory of $\lceil \log_2 6 \rceil = 3$ bits.

Finite State Machines viewed as Graphs

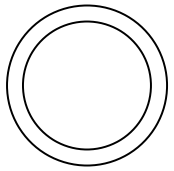
- A state



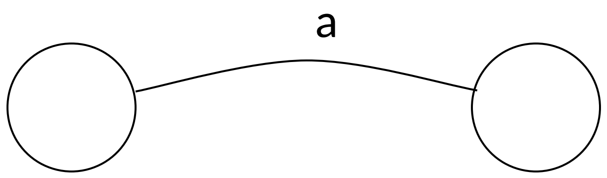
- The start state



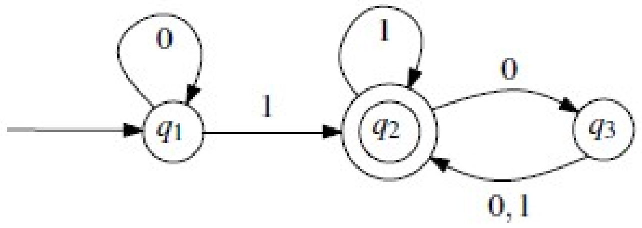
- An accepting state



- A transition



Example 2



- 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 2

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 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.

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

$\Sigma = \{n,d,q\}$

$q_0 = 0$

$F = \{5,6\}$

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

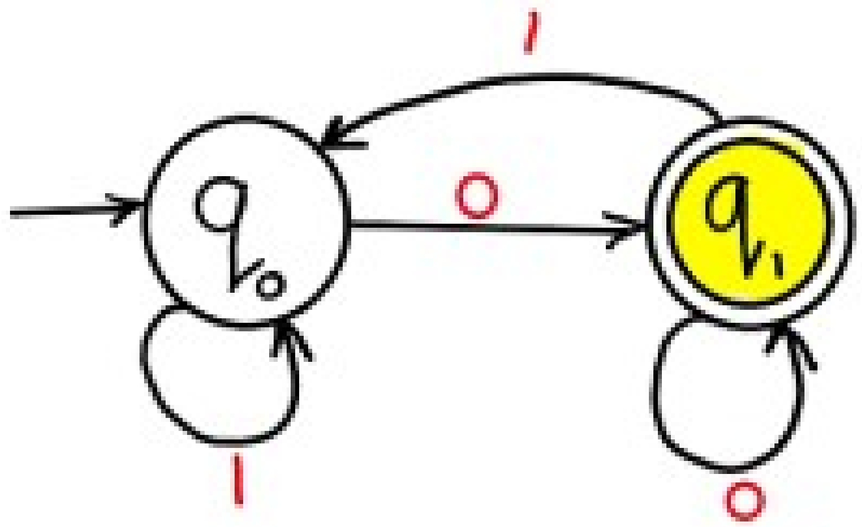
1. the entire string has been read and
2. M is in any of the accept (final) states.

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

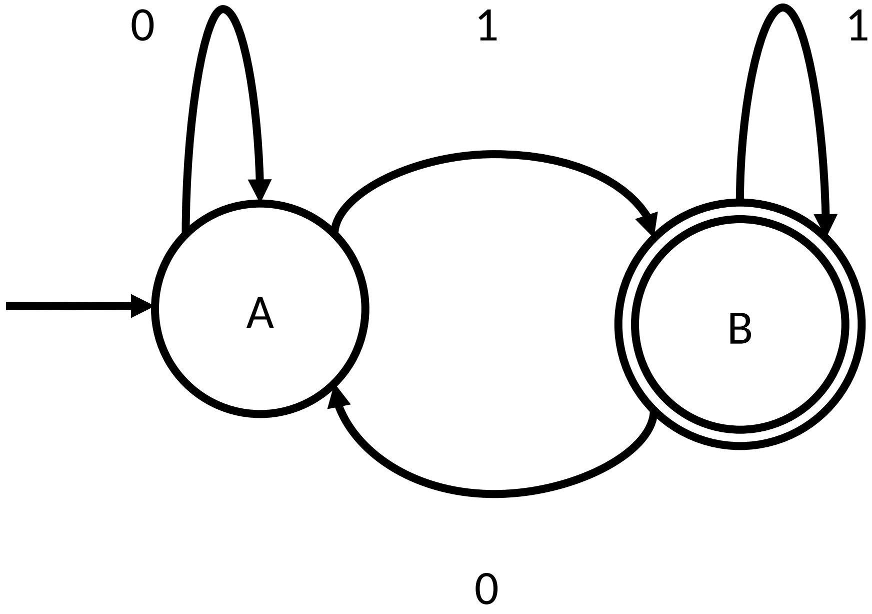
A string ω is accepted by M, iff there is a path from the starting state to any accept state whose edge spells ω.

Example 4

Draw a DFA for the language accepting strings ending with ‘0’ over input alphabets ∑={0, 1} ?

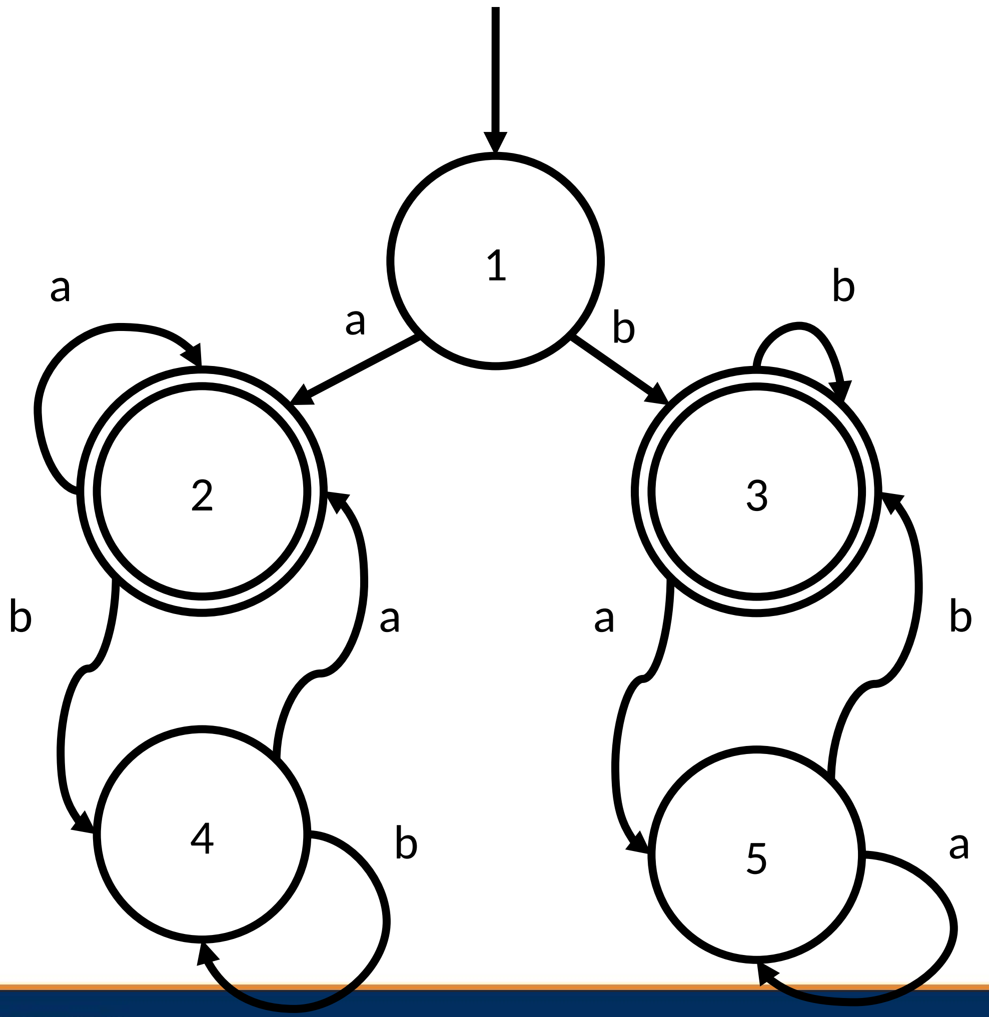


Example 5



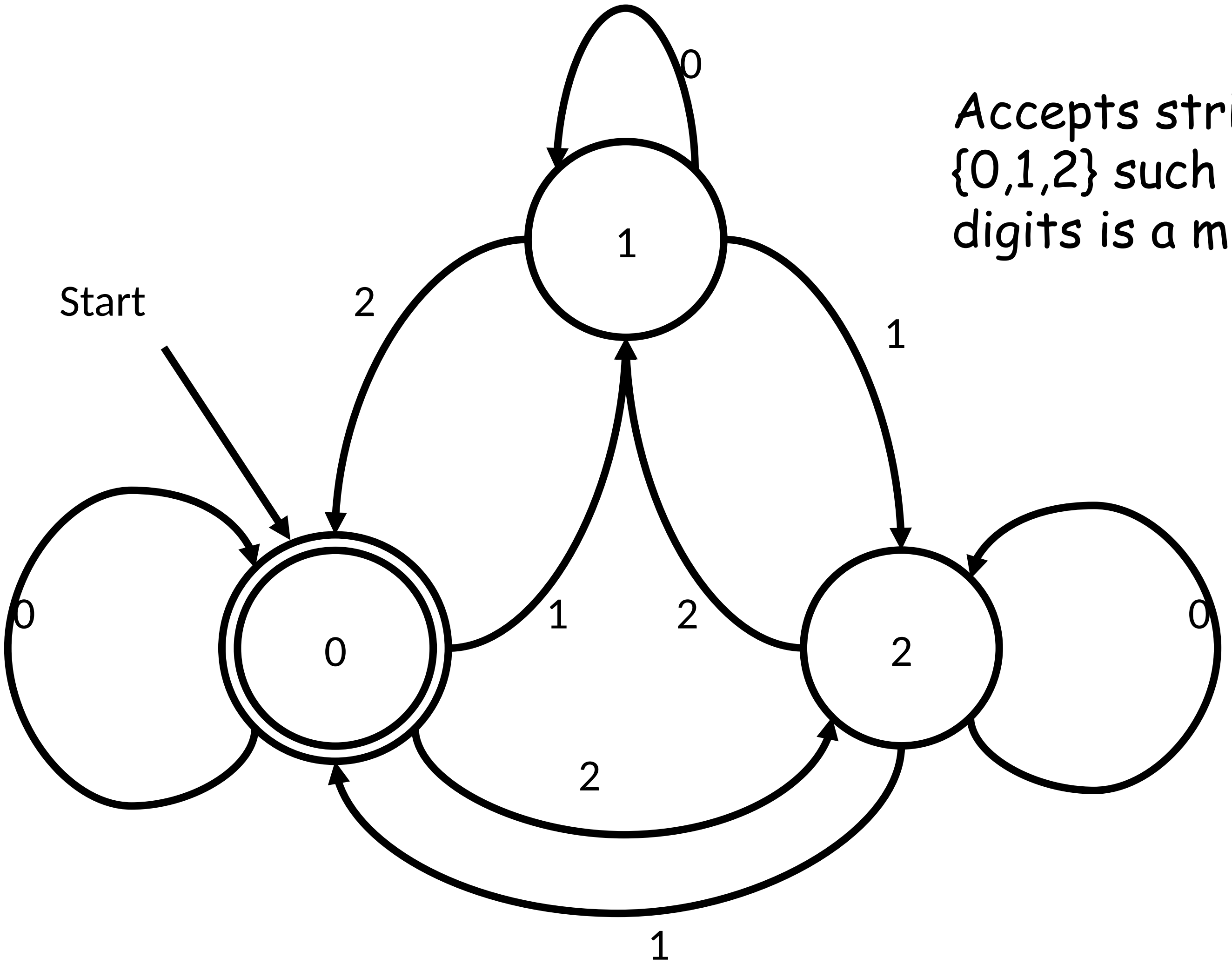
Accepts strings over alphabet {0,1} that end in 1

Example 6



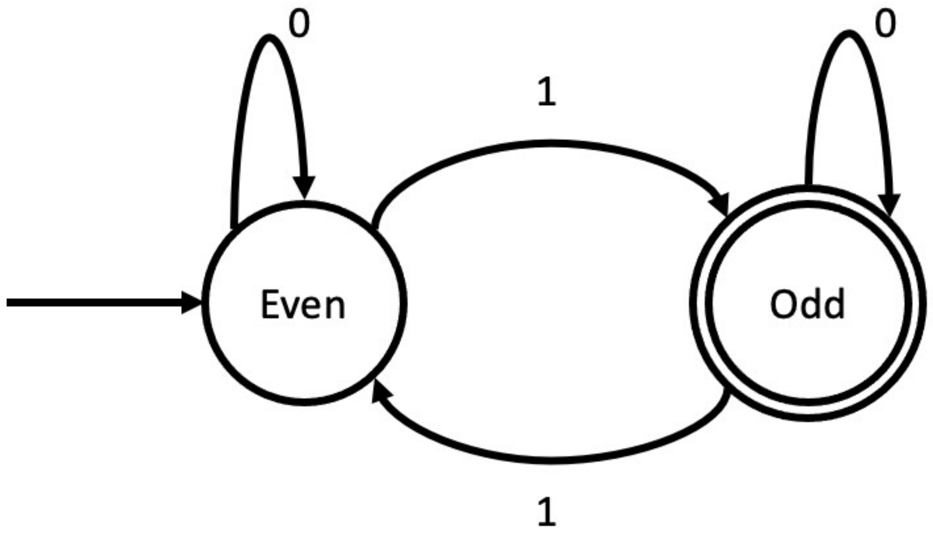
Accepts strings over alphabet {a,b} that begin and end with same symbol

Example 7



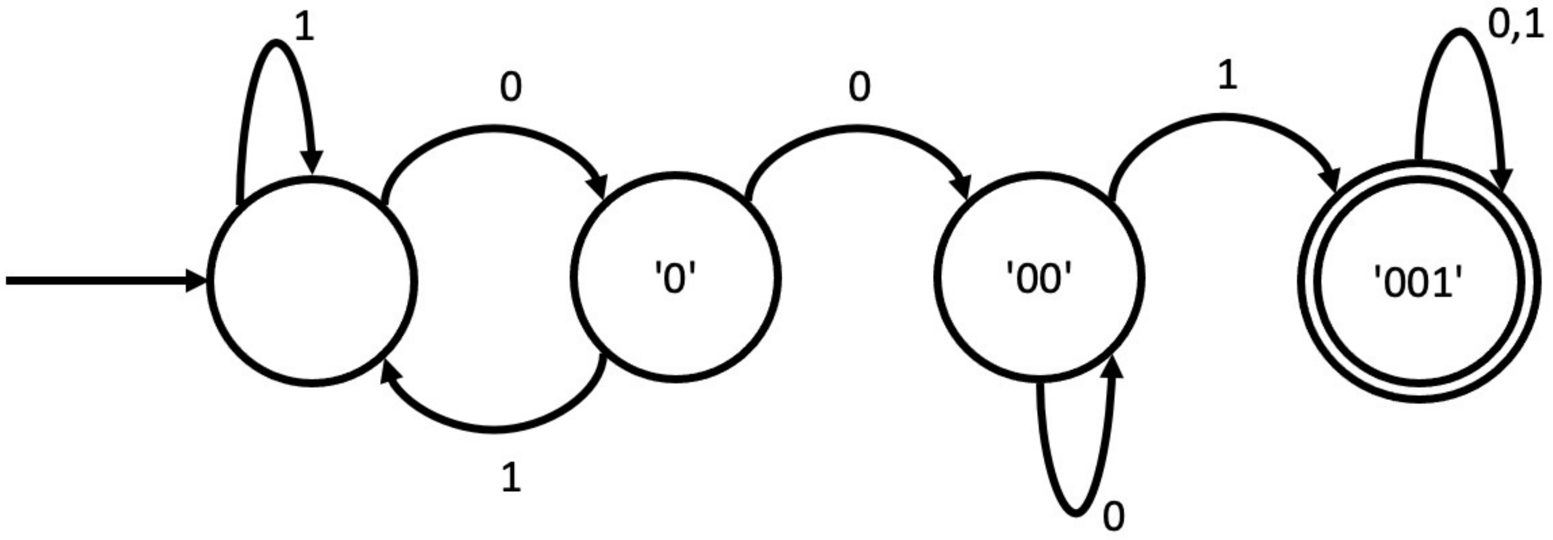
Accepts strings over {0,1,2} such that sum of digits is a multiple of 3

Example 8



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

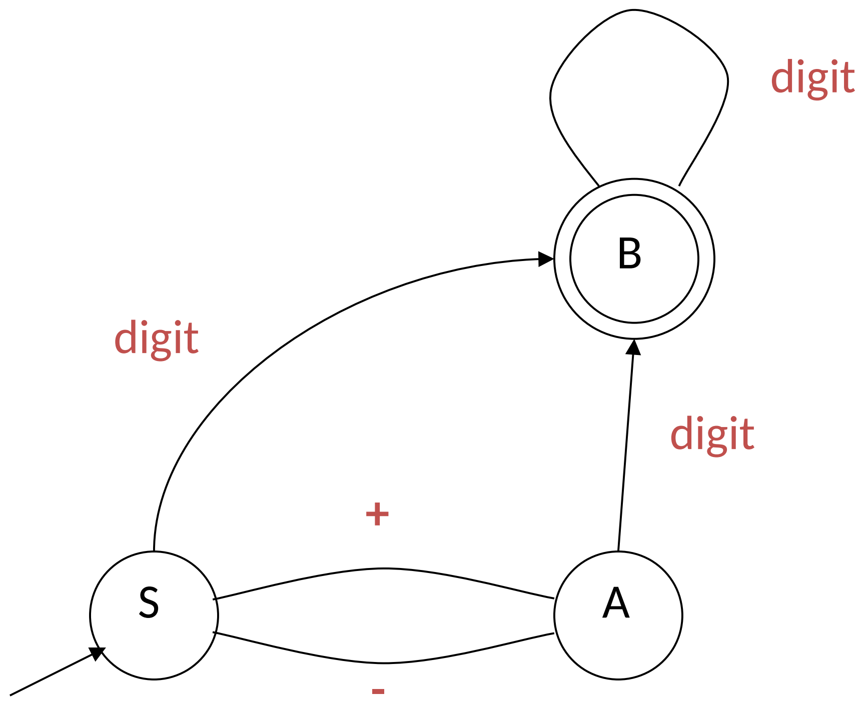
Example 9



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

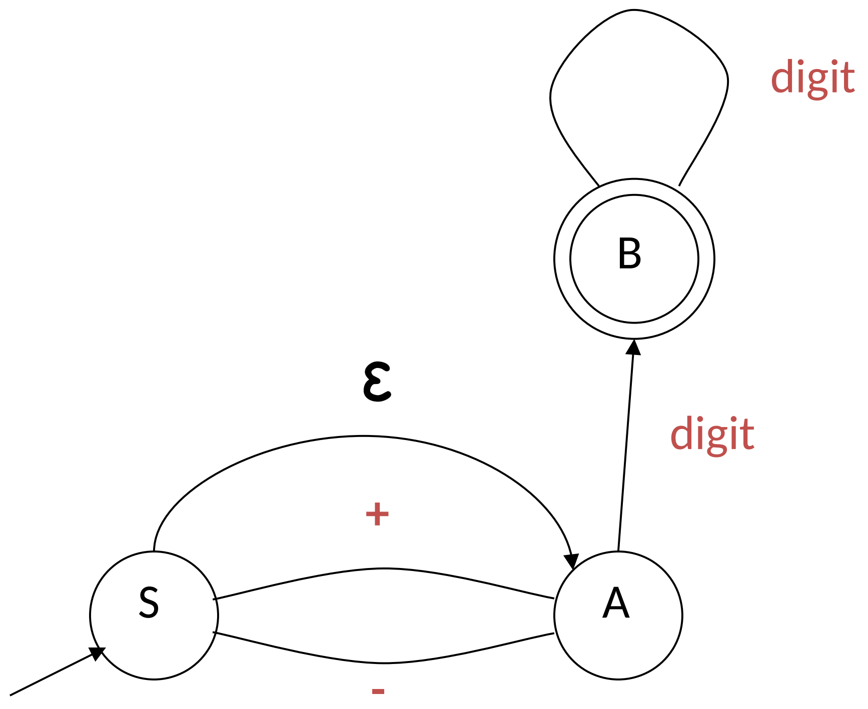
$F \subseteq Q$ is a set of accepting state

N: Q x (Σ U ε) -> P(Q)

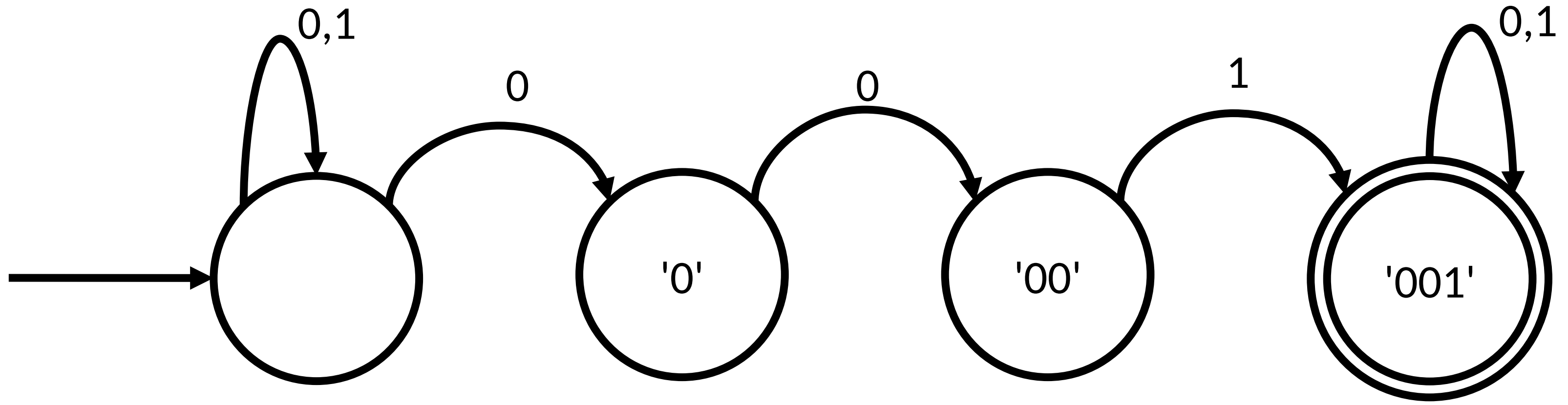
- The symbol ε (epsilon) means no input symbol.
- Input is expanded by epsilon => can go to a different state without reading any input (different from DFSM)
- Can go to multiple states given an input (different from DFSM)

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}
<u>2</u>	{2}	{1,3}
<u>3</u>	{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}
<u>3</u>	{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'*
- Given NFSM M = (Σ, Q, qo, F, N) convert in to a DFSM M' = ((Σ', Q', qo', F', N')

Algorithm

- **Input** − An NDFA
- **Output** − An equivalent DFA
- **Step 1** − Create state table from the given NDFA.
- **Step 2** − Create a blank state table under possible input alphabets for the equivalent DFA.
- **Step 3** − Mark the start state of the DFA by q0 (Same as the NDFA).
- **Step 4** − Find out the combination of States {Q_0, Q_1,... , Q_n} for each possible input alphabet.
- **Step 5** − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.
- **Step 6** − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

Two Cases:

Case 1: Exclude Epsilon transition (2 methods)

Method 1: Powerset Method

$\Sigma' = \Sigma$

Q' = the powerset of states in M in [ ]

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 [ ]

Ex: M = (Σ={a,b), Q ={1,2,3},q0= 1, F={2},

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

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]		
[]		

CALIFORNIA STATE UNIVERSITY FULLERTON

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]
[]	[ ]	[ ]

e.g.,

N’([1], a) = N(1,a) in [ ] = [1,2]

N’([1,3], b) = N(1,b) U N(3,b) in [ ] = [1,3]

N’([1,2,3], a) = N(1,a) U N(2,b) U N(3,b) in [ ] = [1,2,3]

Observation:

1. Impractical due to the powerset
|Q| = 5 => M' has 32 states!!
what if |Q| = 10?

2. Some states are unnecessary ex. [ ], [2,] => can never be reached from starting state

Method 2: Subset Method (A more practical Method)

The idea is: Include in N', only states that are needed

```
M' = (  Σ' = Σ, $q_{0'} = [q_0]$
N' :  Start with q0
    While incomplete rows remaining do
        Begin
        let x = [p1,p2..pn] be the current state
        For each alphabet a in Σ' do
            Begin
            Find y = N'(x,a)
            Add y to the table
             If  y not in N' then add y to N' as a new state
             End
        End
Q' = the state in N'
F' = all states in Q' that contains at least one accepting state
in M )
```

Ex: M is given as follows:

		a	b
q0=	1	{1,2}	{1}
	$\underline{2}$	{2}	{1,3}
	3	{2,3}	{1,3}

M' = (Σ'={a,b},

q0'= [1],

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

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

F'={[1,2],[1,2,3]})

Could SIMPLIFY M'

	a	b
A	B	A
$\underline{B}$	B	C
C	D	C
$\underline{D}$	D	C

Case 2: Include Epsilon transition

First Def: The ε−Closure of a state q ∈ Q is the set of all states that can be reached from q by means of epsilon transition including q.

Ex.	a	b	ε
1	{1}	{2}	{2,3}
2	{2}	{3}	{ }
3	{2}	{1}	{2}

ε−Closure(1) = {2,3,1},
ε−Closure(2)={2},
ε−Closure(3)={2,3}

We need to do 2 modifications to method 2 (subset method):

➢ Starting State of M'   => Find ε–Closure of q0

➢For each new state [p1,p2,…pn], we must include all states that can be reached by means of ε transitions
⇒Find ε–Closure of ([p1,p2,…,pn])
⇒which is ε–Closure(p1) U ε–Closure(p2) U ….. ε–Closure(pn)

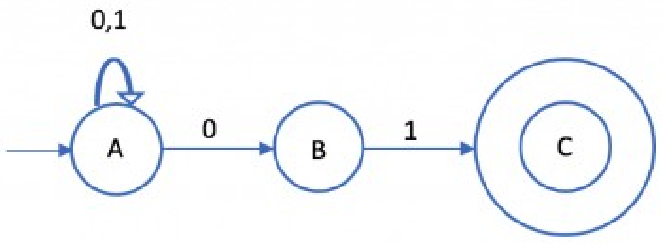
Example 2

Given below is the NFA for a language L= {Set of all string over (0, 1) that ends with 01 } construct its equivalent DFA.

NFA

State Transition Diagram for NFA

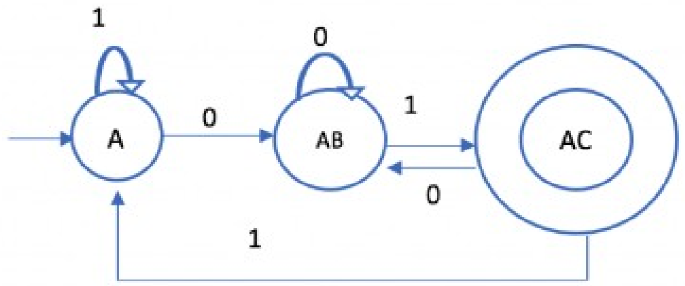
	0	1
→ A	{A, B}	{A }
B	Ø	{C}
* C	Ø	Ø



DFA

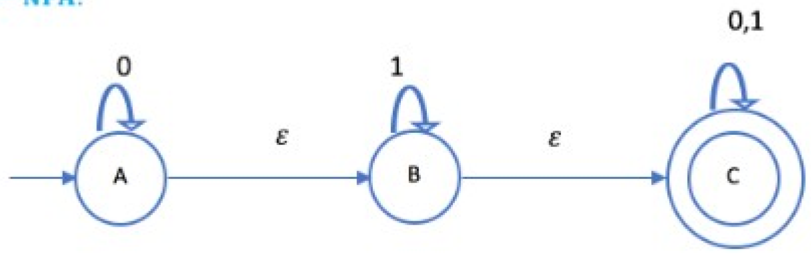
State Transition Diagram for DFA

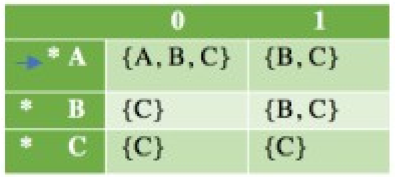
	0	1
→ A	AB	A
AB	AB	AC
* AC	AB	A



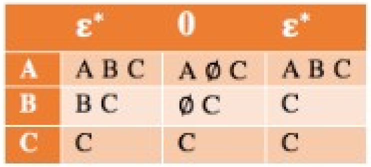
Epsilon NFA to NFA

ε – NFA:

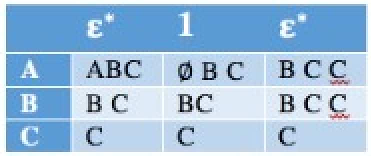




	0	1
→* A	{A, B, C}	{B, C}
* B	{C}	{B, C}
* C	{C}	{C}

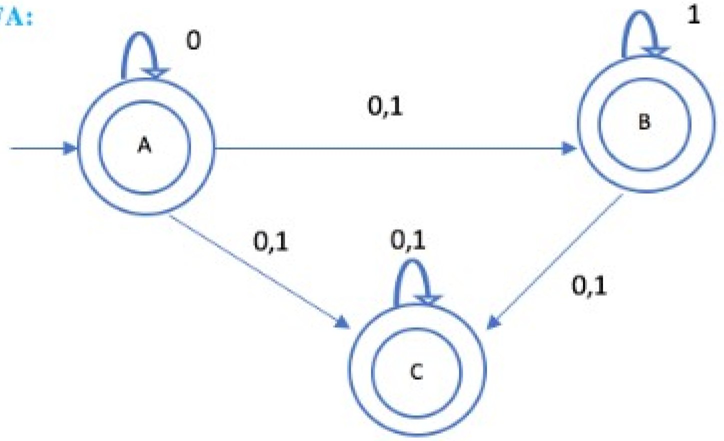


	ε*	0	ε*
A	A B C	A ∅ C	A B C
B	B C	∅ C	C
C	C	C	C



	ε*	1	ε*
A	ABC	∅ B C	B C C
B	B C	BC	B C C
C	C	C	C

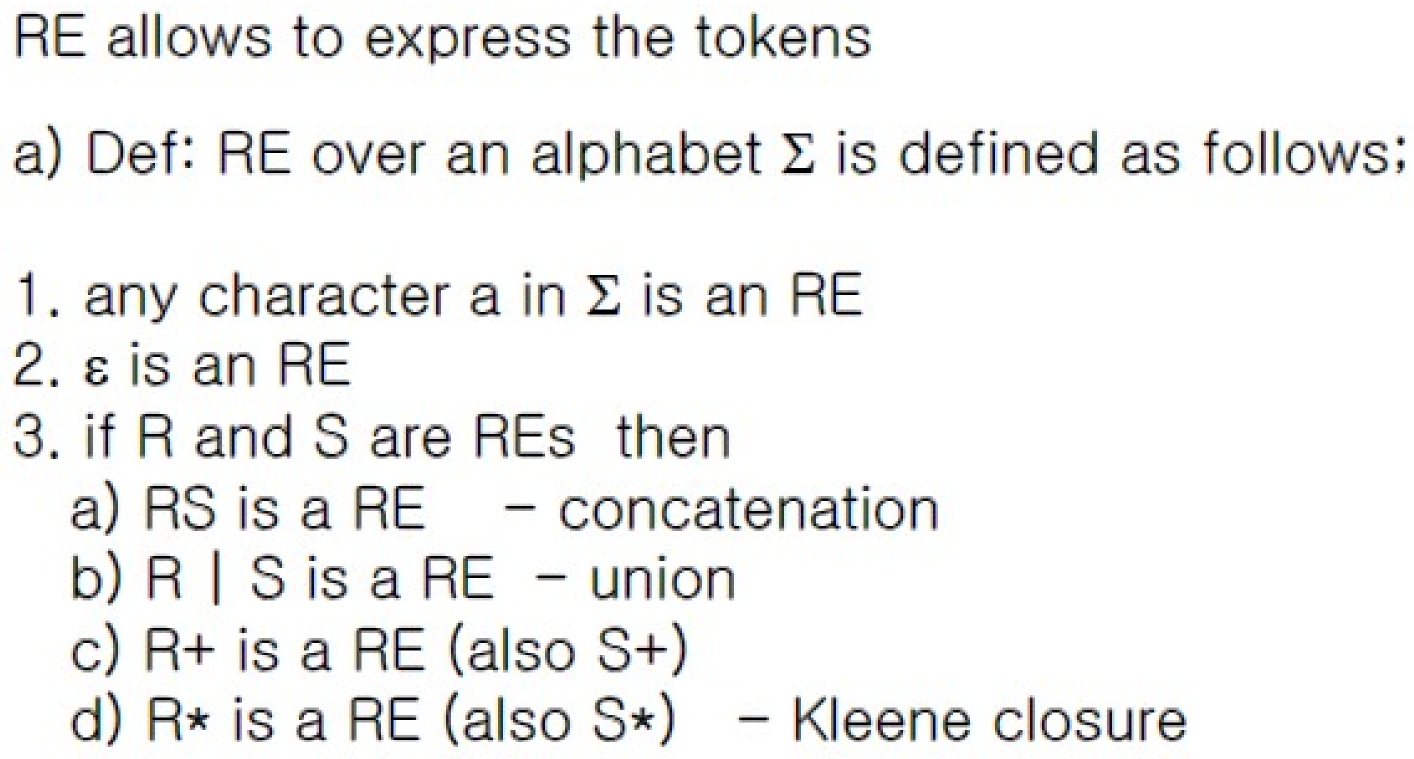
NFA:



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

contd.



RE allows to express the tokens

a) Def: RE over an alphabet Σ is defined as follows:

1. any character a in Σ is an RE
2. ε is an RE
3. if R and S are REs then
   a) RS is a RE – concatenation
   b) R | S is a RE – union
   c) R+ is a RE (also S+)
   d) R* is a RE (also S*) – Kleene closure

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

$(0U1)^*$

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

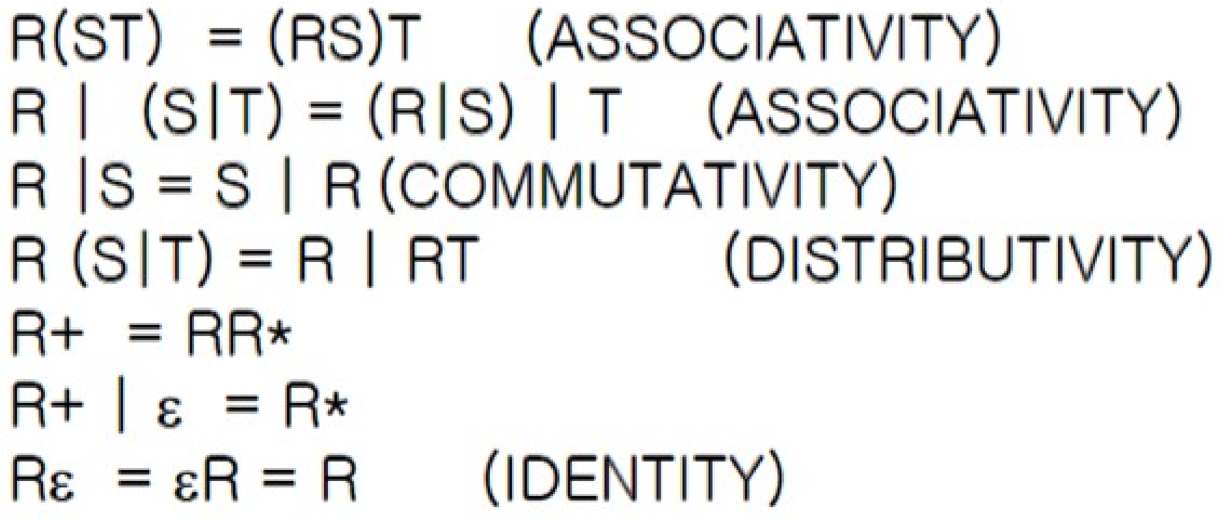
➢ A more interesting examples:

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

l (l | d | _) * −> an identifier (other definitions of id)

=> We can describe the tokens in terms of RE

Equivalence Rules for RE



R(ST) = (RS)T (ASSOCIATIVITY)

R | (S|T) = (R|S) | T (ASSOCIATIVITY)

R |S = S | R (COMMUTATIVITY)

R (S|T) = R | RT (DISTRIBUTIVITY)

R+ = RR*

R+ | ε = R*

Rε = εR = R (IDENTITY)

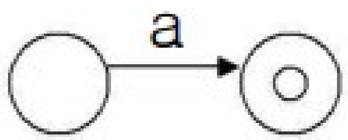
2.6 FSMs and REs

- 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

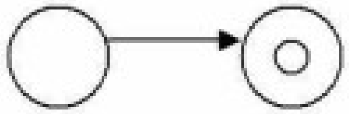
Equivalence between FSMs and REs

- Theorem: There exists an FSM for each RE and vice versa.
- Q: How can we construct an FSM for a given RE?
- A: Using Thompson Construction Method

1. M that recognizes a $\in \Sigma$

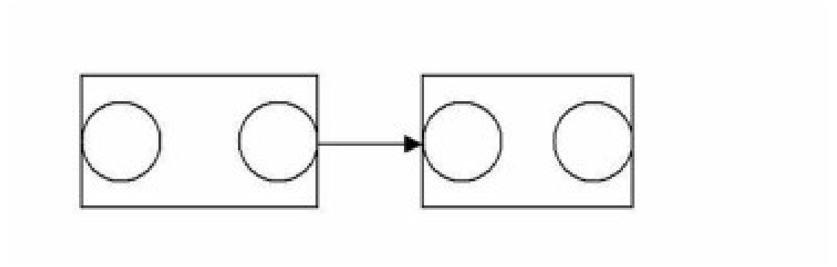


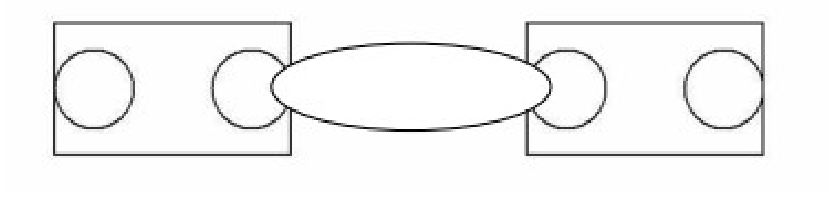
2. M that recognizes ε



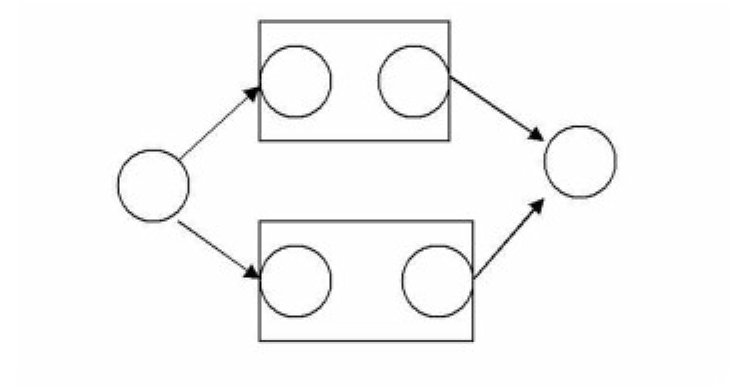
Edges not marked are ε

3. if R and S are REs then
a) RS is a RE – concatenation

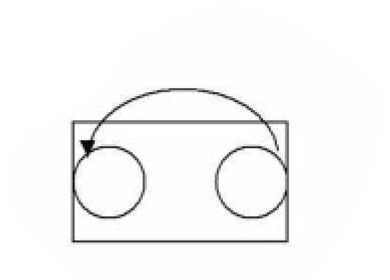




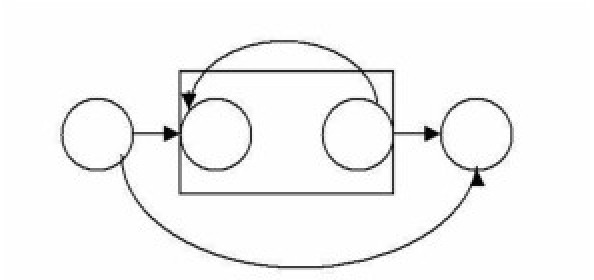
b) R | S is a RE – union



c) R+



d) R* – Kleene Closure



Example 1

Describe the following sets as regular expressions –

{1,11,111,1111,11111……} = 1^+

{^,0,00,000,0000……..} = 0^*

Example 2

Language over (a,b) accepting strings of length exactly 2

L = {aa,ab,ba,bb}

R = aa + ab + ba + bb

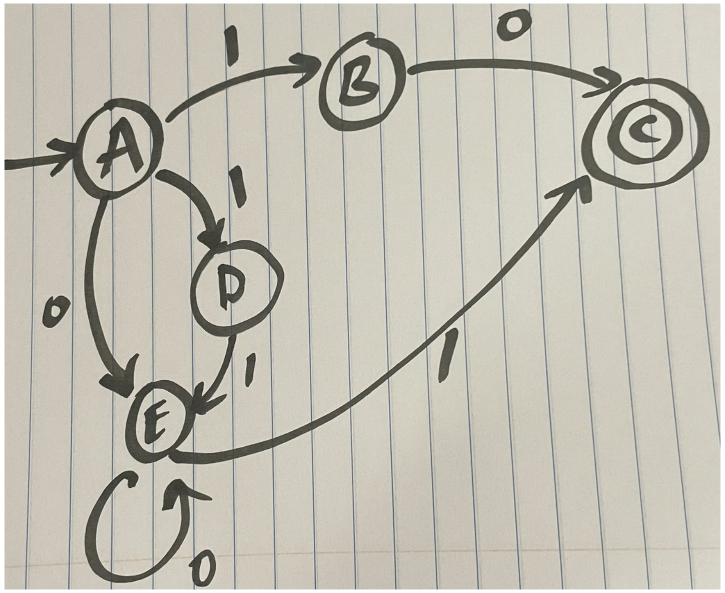
  = a(a+b)+ b(a+b)

  = (a+b)(a+b)

Example 3

Convert the following regular expressions to FA

a) 10 + (0+11) 0*1



Example 4

Convert the following regular expressions to FA (Try yourself)

a) b a*b

b) (a+b) c

c) a(bc)*

$$\begin{tikzcd}
\text{DFSM} \arrow[r, shift left] \arrow[d, leftrightarrow, "\text{DEF}"'] & \text{NFSM} \arrow[l, shift left] \arrow[d, shift left] \\
\text{Regular Language} & \text{Regular Expression} \arrow[u, shift left]
\end{tikzcd}$$

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

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

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)

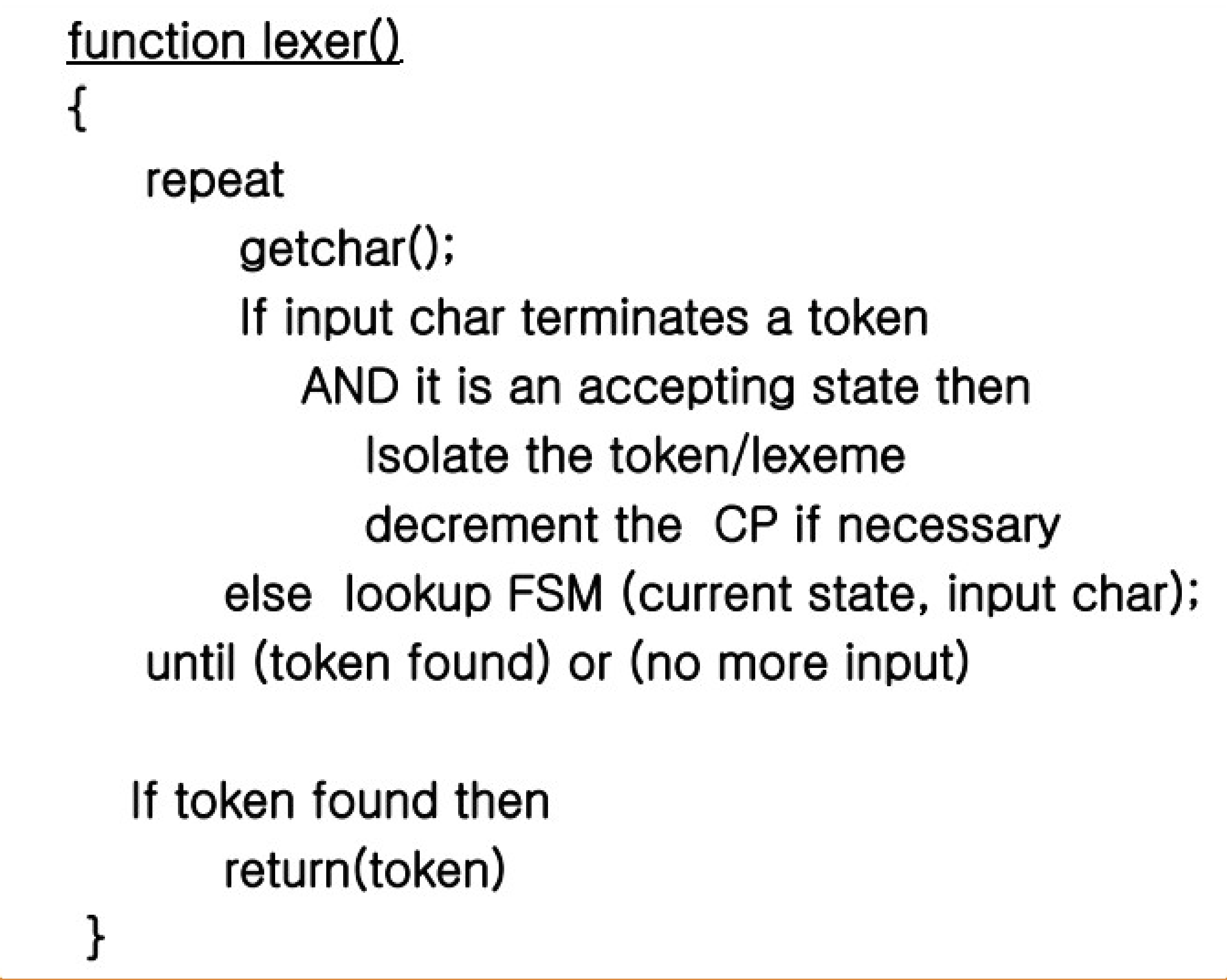
Step1) RE for Identifier is l(l | d) * and RE for an integer is d+

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

Implementation of a Lexer



```
function lexer()
{
    repeat
        getchar();
        If input char terminates a token
            AND it is an accepting state then
                Isolate the token/lexeme
                decrement the  CP if necessary
        else  lookup FSM (current state, input char);
    until (token found) or (no more input)

   If token found then
        return(token)
 }
```