Finite-State Machines

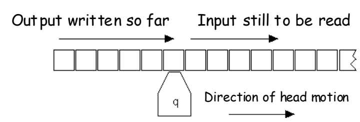
State Machines in General

- "Mathematical" (as opposed to mechanical) machines
  - Turing Machines (potentially infinite-state)
  - Finite-state machines
  - Other categories (cf. CS 142, Theory of Computation)

What are Finite-State Machines?

- A primitive computational model, related to many facets of computing:
  - A severely-restricted type of Turing machine
  - Model of switching circuits with ***memory***
  - The building blocks for most real-life computers
  - Parsing for a limited family of languages (called "regular" languages or finite-state languages)
  - Regular expressions: used for textual pattern matching
  - Real-time software applications

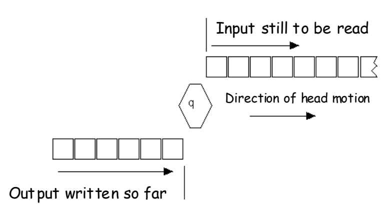
FSM as a "crippled" TM



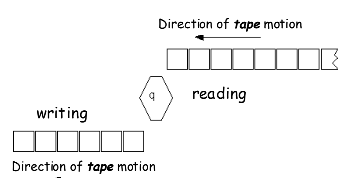
Can move in one direction only;

Symbol written is never again changed.

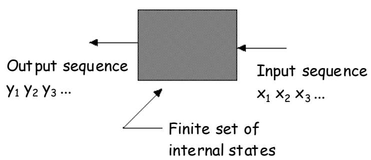
FSM with separate I/O



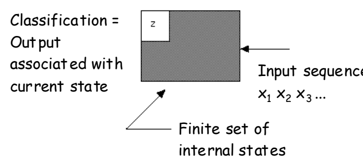
FSM as head-stationary, tape-moving, device



FSM as Sequence *Transducer*



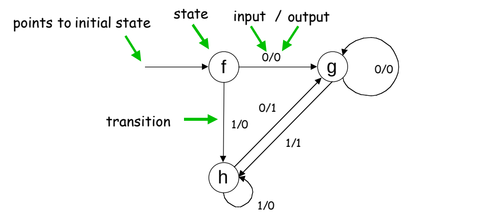
FSM as a Sequence ***Classifier***



Edge-Detector Example

input	output
0	0
00	00
01	01
011	010
0111	0100
01110	01001
"edges"	detection of edges

Transducer: An Edge Detector



The state is recording the previous input.

Whenever the current input differs, an edge has been detected.

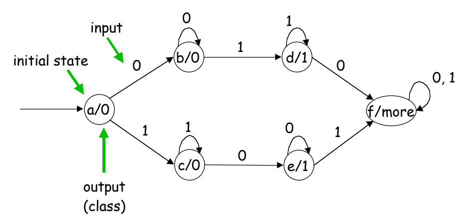
The first input is not considered an edge.

Transducer Transcribed to a rex Program

```
edgeDetector(input) = f(input);

f([]) => [];
f([0 | remainder]) => [0 | g(remainder)];
f([1 | remainder]) => [0 | h(remainder)];

g([]) => [];
g([0 | remainder]) => [0 | g(remainder)];
g([1 | remainder]) => [1 | h(remainder)];

h([]) => [];
h([0 | remainder]) => [1 | g(remainder)];
h([1 | remainder]) => [0 | h(remainder)];

test(edgeDetector([0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1]),
                 [0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0]);
```

A related Classifier:

How many **edges** were there so far (0, 1, more)



Classifier Transcribed to a rex Program

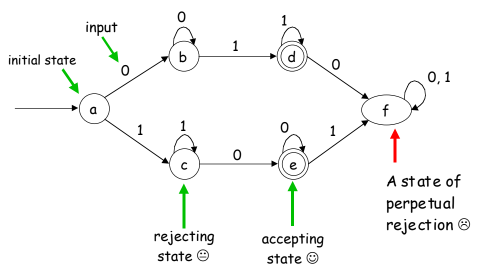
```
edgeClassifier(input) = a(input);

a([]) => 0;
a([0 | remainder]) => b(remainder);
a([1 | remainder]) => c(remainder);

b([]) => 0;
b([0 | remainder]) => b(remainder);
b([1 | remainder]) => d(remainder);

c([]) => 0;
c([0 | remainder]) => e(remainder);
c([1 | remainder]) => c(remainder);

d([]) => 1;
d([0 | remainder]) => f(remainder);
d([1 | remainder]) => d(remainder);

e([]) => 1;
e([0 | remainder]) => e(remainder);
e([1 | remainder]) => f(remainder);

f([]) => "more";
f([0 | remainder]) => f(remainder);
f([1 | remainder]) => f(remainder);
```

Finite-State Acceptors (FSAs)

- **Acceptors** are Classifiers with only **2 classes**: accepted and rejected.
- Rejection is not necessarily final: additional input can convert to accepted.
- Typically acceptors show accepting states with **double outline**, rejecting states have single outline.

A related Acceptor:

Accept sequences with exactly one edge



Acceptor Transcribed to a rex Program

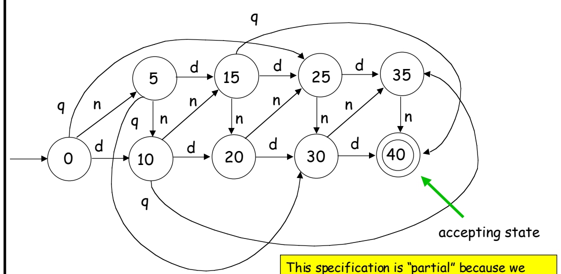
Conversions

- Every classifier can be represented as a "gang" of acceptors, by encoding the class.
- Every transducer can be represented as an "equivalent" classifier.
- Therefore, studying acceptors, the simplest model, yields insight for all finite-state machines.

What can an Acceptor Do?

- The Pepsi Machine near B101.
- Coins of 5, 10, and 25 cents can be entered (referred to by input symbols **n**, **d**, **q**, respectively).
- Accepts when a total of 40 cents (or more ☺) has been entered.

Pepsi Acceptor (partial)



This specification is "partial" because we have not said what happens when q is input to states 20, 25, 30, 35, etc.

Types of Acceptor Problems

- **Analysis**: What does a given acceptor do (e.g. in English)?
- **Synthesis**: Construct an acceptor that performs according to a specification.
- **Realization**: Show the logic for a switching circuit that realizes an acceptor.
- **Abstraction** ("reverse engineering"): From a switching circuit, give the corresponding acceptor.

Languages for FSAs

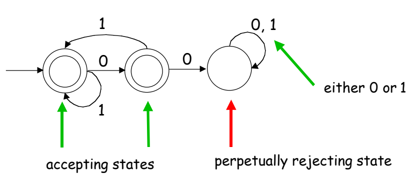
- A convenient way to characterize an acceptor is by its **language**, the set of all input sequences it accepts.
- Typically the language will be infinite, although there are also cases of finite languages.

Language Examples

- The set of all strings over {0, 1} such that every 0, if followed by any symbol, is followed by a 1.
- The set of all strings over {0, 1} such that the number of symbols is a multiple of 4.
- The set of all binary numerals that, m.s.b. first, are multiples of 3:
  {0, 11, 110, 1001, 1100, 1111, …}
  (corresponding to 0, 3, 6, 9, 12, 15, …)

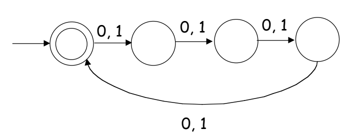
Language Examples

- The set of all strings over {0, 1} such that every 0, if followed by any symbol, is followed by a 1.



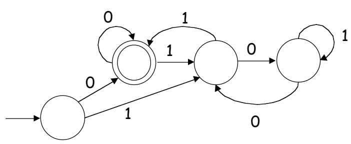
Language Examples

- The set of all strings over {0, 1} such that the number of symbols is a multiple of 4.



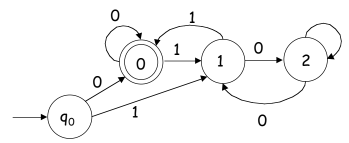
Language Examples

- The set of all binary numerals that, m.s.b. first, are multiples of 3:
  {0, 11, 110, 1001, 1100, 1111, …}
  (corresponding to 0, 3, 6, 9, 12, 15, …)



Correctness of the Multiples-of-3 Example

Let n be the numeral input so far. For every n, there is a k and an r < 3, such that n = 3k+r (r = n%3). States, other than the starting state, are identified with with r.

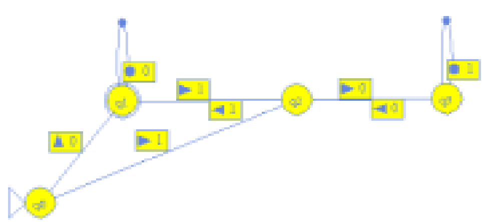


Inputting a 0 takes n to 2n, and
inputting a 1 takes n to 2n+1.
So inputting a 0 takes 3k+r to 6k+2r, while
Inputting a 1 takes 3k+r to 6k+2r+1.

r	(2r) % 3	(2r+1) % 3
0	0	1
1	2	0
2	1	2

Acceptor as Entered in JFLAP

(Applet-based tool by Susan Rodger:
http://www.cs.duke.edu/~rodger/tools/jflap)

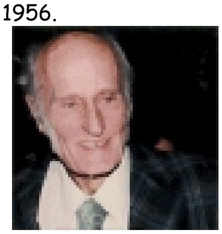


Characterization of Finite-State Machines by "Regular Expressions"

- **Regular expressions** are a *machine-independent* way of specifying a language.
- They are often used in textual **pattern-matching** applications.
- They are closely related to **grammars**, but the form of recursion is limited to "iterative" forms only.

Regular Expressions

- Discovered by the mathematical-logician S.C. Kleene (1909-1994, Prof. at U. of Wisconsin) in studying "nerve nets" in 1956.
- Kleene was also a principal developer of the field of recursion (computability) theory



About Mr. Kleene

Kleene pronounced his last name /klay'nee/.

/klee'nee/ and /kleen/ are extremely common mispronunciations. His first name is /steev'n/, not /stef'n/.

His son, Ken Kleene <kenneth.kleene@umb.edu>, wrote: "As far as I am aware this pronunciation is incorrect in all known languages. I believe that this novel pronunciation was invented by my father."

Regular Expressions Defined

- A regular expression (RE) is always defined with respect to a finite **alphabet** of symbols, Σ. The definition is inductive:
- Basis:
  - Any symbol in Σ is an RE.
  - The special symbol λ is an RE (often ε is used instead of λ).
  - The special symbol ∅ is an RE.
- Induction step: If R and S are RE's, then so are:
  - RS
  - R | S
  - R*

Regular Expression Examples

- Take Σ = {0, 1}.
- Basis:
  - Any symbol in Σ is an RE: 0 1
  - The special symbol λ is an RE: λ
  - The special symbol ∅ is an RE: ∅
- Induction step: If R and S are RE's, then so are:
  - RS: 00 01 0001 1010 1(00 | 11)*0
  - R | S 00 | 11 0 | 1 | λ
  - R* 0* 01*0 (00 | 11)*

Meaning of Regular Expressions(1)

- Each regular expression R denotes a **language** (set of strings) L(R) over its alphabet:
- Basis:
  - A symbol σ in Σ denotes the language of one string of one letter: L(σ) = {σ}.
  - The special symbol λ denotes the empty string (no letters): L(λ) = {λ}.
  - The special symbol ∅ denotes the empty set (no strings): L(∅) = ∅

Meaning of Regular Expressions (2)

- Induction step: Suppose R and S are regular expressions and L(R) and L(S) have been defined. Then (concatenation of two strings)
  - L(RS) = {xy | x ∈ L(R) and y ∈ L(S)}
  - L(R | S) = L(R)∪ L(S)
  - $L(R^*) = \{\lambda\} \cup L(R) \cup L^2(R) \cup L^3(R)$ ...

  where $L^k(R)$ means the language formed by concatenating k strings, each one from L(R).

Similarity to Grammar Rules

Suppose that we have a grammar in which
auxiliary symbol r derives the strings in L(R) and
auxiliary symbol s derives the strings in L(S).
Then:

- Adding t → r s would make t derive the strings in L(RS).
- Adding t → r | s would make t derive the strings in L(R | S).
- Adding t → {r} would make t derive the strings L(R*).

Note on Precedence in Regular Expressions

- It is common to omit parentheses.
- The binding order is:
  - * binds most tightly
  - juxtaposition is next
  - | binds most weakly

Examples of RE's, with Meanings

- 0101
  The set of one string "0101".
- 0101 | 1010
  The set of two strings, "0101" and "1010".
- 1(0101 | 1010)0
  The set of two strings, "101010" and "110100".
- 01*0
  The set of strings that begin and end with 0 and contain a continuous run of 1's (of length 0 or more).

Examples of RE's, with Meanings

- 0*1*
  The set of strings in which no 1 is followed by a 0.
- 0*1*0*1*
  The set of strings in which at most one 1 is followed by a 0.
- 0*(100*)*
  The set of strings in which every one is followed by a 0.

Try These

- (0*10*1)*0*
- ((0 | 1)(0 | 1))*
- 0*10* | 1*01*
- (0*1*)*

Give Regular Expressions (over alphabet {0, 1}) for

- The set of strings with at most two 0's
- The set of strings with more than two 0's
- The set of strings in which 0's and 1's strictly alternate

Kleene's Remarkable Result

- The languages accepted by finite-state acceptors and the languages denoted by regular expressions are the same thing.

In other words:

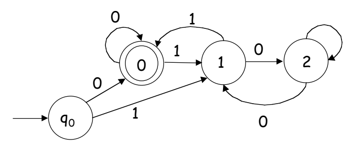
- Part I: The language accepted by any finite-state acceptor can be expressed as a regular expression.
- Part II: For every regular expression, there is a finite state acceptor that accepts the language denoted by the expression.

Proof of Part I

- The language accepted by any finite-state acceptor can be expressed as a regular expression.

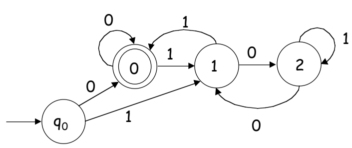
Proof of Part I:

- Given a finite-state acceptor, how to derive a regular expression?
- Example (multiples of 3):



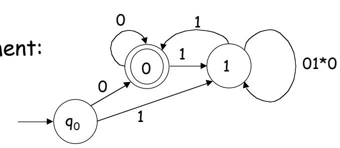
Idea of Part I: (analogous to Gaussian elimination)

- Eliminate states, replacing paths with regular expressions that represent those paths.
- Original:



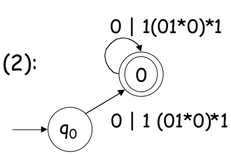
Idea of Part I (2)

- Replacement:



Idea of Part I (3)

- Replacement (2):

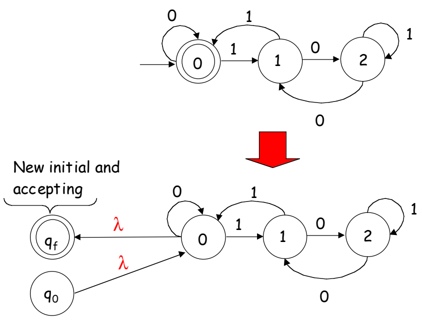


- Final: (0 | 1 (01*0)*1) (0 | 1(01*0)*1)*

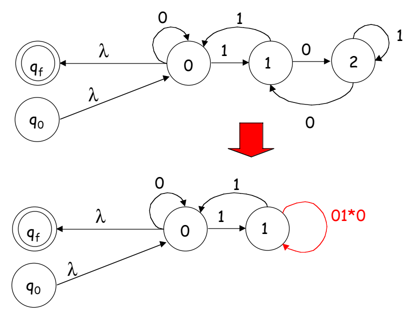
Sanity-Preserving Technique for Elimination

- This helps deal with cases in which initial and accepting states overlap or are involved in loops.
- Introduce new states for initial and accepting.
- Connect new initial state to original initial state by λ transition.
- Connect all accepting states to a single new accepting state via λ transitions.
- The original initial and accepting states are now ordinary states.
- Eliminate, in succession, all nodes other than the new initial and accepting states.
- The regular expression for the acceptor is the one connecting initial to accepting .

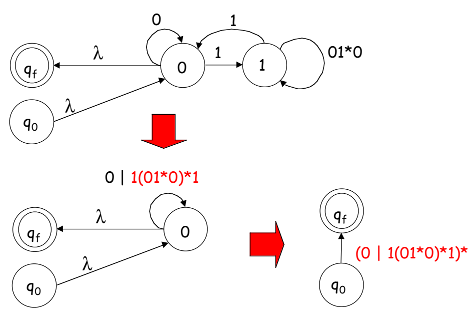
Sanity-Preservation Illustrated



Elimination, with sanity preservation



Elimination, with sanity preservation



Proof of Part II

- For every regular expression R, there is an FSA that accepts L(R), the language denoted by R.

Non-Deterministic FSAs

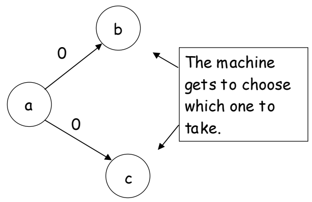
- The easiest way to prove part II is to appeal to the idea of a non-deterministic finite-state acceptor (NFSA):
  - Part IIa: For every regular expression R, there is an NFSA that accepts L(R).
  - Part IIb: For every NFSA N there is a (deterministic) finite-state acceptor that accepts L(N).

Non-Deterministic FSAs

- A non-deterministic finite-state acceptor (NFSA) is a finite-state acceptor with free-choice of transitions:
  - A given state may have more than one transition leaving with the same symbol, or
  - A state may be left spontaneously via a λ transition.

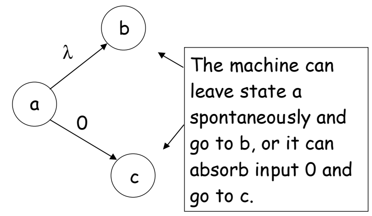
Non-Deterministic FSAs

- A given state may have more than one (or even no) transition leaving with a given symbol.



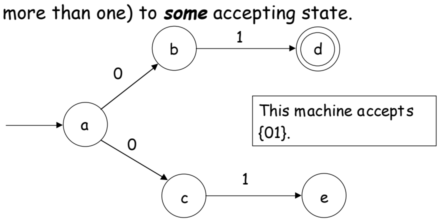
Non-Deterministic FSAs

- A state may be left spontaneously via a λ transition.



Acceptance Notion for NFSAs

- An NFSA accepts an input sequence iff there is ***some*** path from ***some*** initial state (an NFSA can have more than one) to ***some*** accepting state.



Proof of Part IIa

- Part IIa: For every regular expression R, there is an NFSA that accepts L(R).
- This proof is by **structural induction** on the formation of regular expressions.
  - Basis:
    - Any symbol in Σ is an RE.
    - The special symbol λ is an RE.
    - The special symbol ∅ is an RE.
  - Induction step: If R and S are RE's, then so are:
    - RS
    - R | S
    - R*

Proof of Part IIa (1)

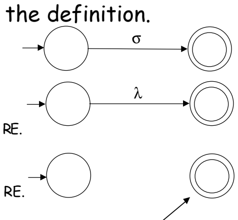
- We construct an accepting NFSA for each RE introduced in the definition.
  - Basis:
    - Any symbol in Σ is an RE.

This is a string, not an alphabet symbol.

- The special symbol λ is an RE.

This is neither a string nor an alphabet symbol.

- The special symbol ∅ is an RE.

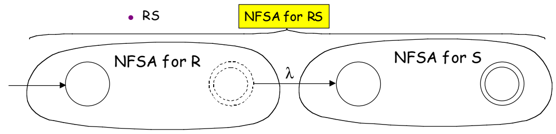


You can't get here from there.

Proof of Part IIa (2)

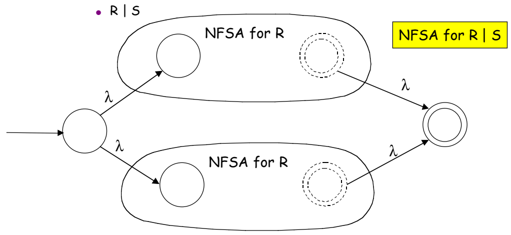
- We construct an accepting NFSA for each RE introduced in the definition.
  - Induction step: If R and S are RE's, then so are:
    - RS
    - R | S
    - R*
  - We assume that NFSA's exist for R and S, and construct them for these three cases:
    - RS

NFSA for RS



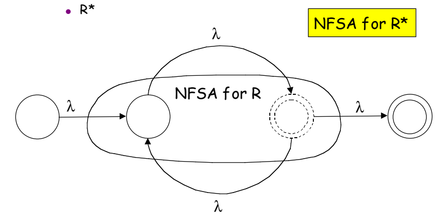
Proof of Part IIa (3)

- We assume that NFSA's exist for R and S, and construct them for these three cases:
  - R | S



Proof of Part IIa (4)

- We assume that NFSA's exist for R and S, and construct them for these three cases:
  - R*



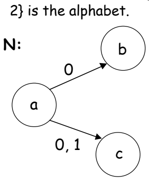
Proof of Part IIb (1)

- For every NFSA N there is a (deterministic) FSA that accepts L(N).
- The idea is that for an NFSA N we can construct a FSA D accepting L(N) by "simulating in parallel" all the **choices** the NFSA could make. An input sequence is accepted iff **any** of those choices led to acceptance in N.

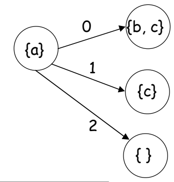
Proof of Part IIb (2)

- To simulate an NFSA, we construct D to have as its states **subsets** of the states of N. The transitions of D emulate all transitions for N "in parallel". For example, suppose that {0, 1, 2} is the alphabet.

In N:



In D:

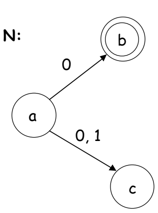


Definition of f, the state transition for D:
$f(S, \sigma) = \{q' \mid (\exists q \in S)\ q \xrightarrow{\sigma} q' \text{ in } N\}$

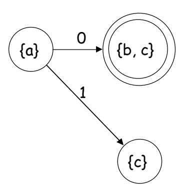
Proof of Part IIb (3)

- An **accepting** state in D is any that has an accepting state of N as a member.

In N:



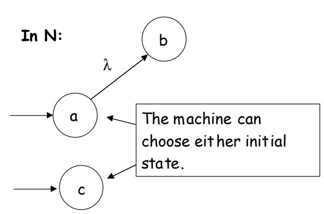
In D:



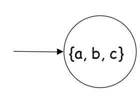
Proof of Part IIb (4)

- The **initial** state in D is the set of all states reachable from ***some*** initial state in N by the empty sequence (i. e. including λ transitions)

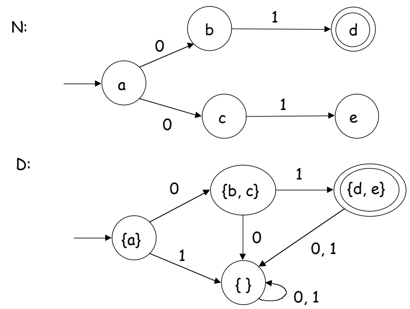
In N:



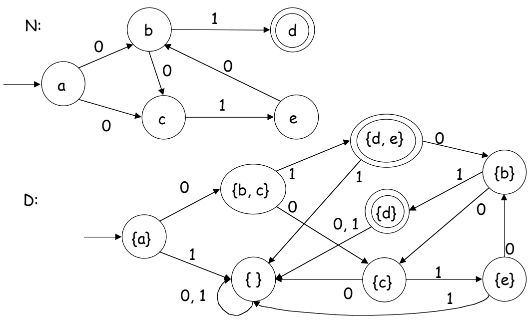
In D:



The Complete Construction for a Simple Example



A More Complex Example with a Loop

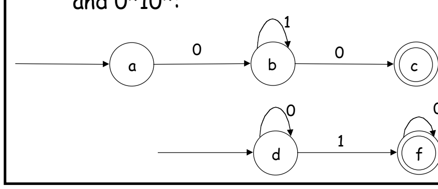


This Completes the Proof of Kleene's Theorem

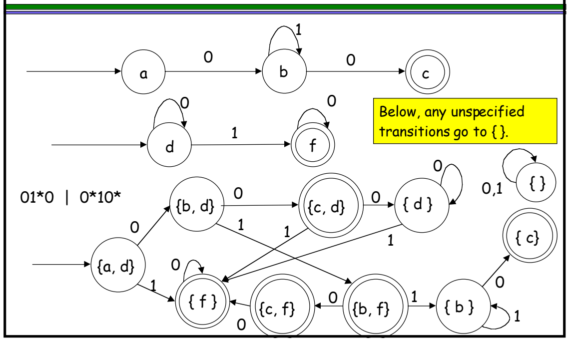
- We now know that the following are equivalent:
  - L is a language denoted by some regular expression.
  - L is a language accepted by an NFSA.
  - L is a language accepted by an FSA.

Example: Regular Expression to FSA (1)

- Construct an FSA for the RE
  01*0 | 0*10*
- By inspection we can do NFSA's for 01*0 and 0*10*:



Example: Regular Expression to FSA (2)



Regular Expressions in Everday Practice:

e.g. Unix **egrep**

used for searching for ***lines containing*** matching strings in files

- Do `man regexp` to get this information on turing:
  - Most single characters match themselves (exceptions: `. * [ ] \ ^ $`)
  - . matches any character, except new-line
  - ^ matches beginning of line (must occur first)
  - $ matches end of line (must occur last)
- Examples:
  - `egrep 'elle'` *filename*
  - `egrep 'll.*ll'` *filename* .* is like Σ*
  - `egrep 'll$'` *filename*
  - `egrep '^LL'` *filename*
  - `egrep 'aa|bb|cc'` *filename*
  - `egrep '(aa|bb)c'` *filename*