CS 361 Meeting 3 — 2/12/20

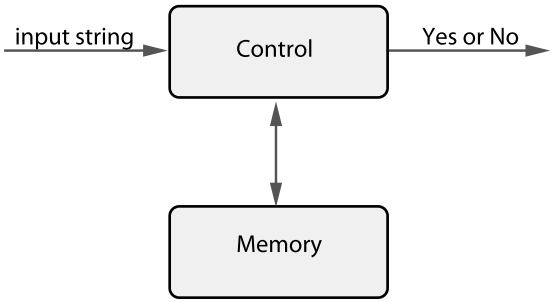
Announcements

1. Homework 1 is due Today.
2. My Hours:
   - Mon., Weds. : 1:30-3:00
   - Thurs.: 1:30-3:30
   - Fri.: 1:30-2:30
3. TA Hours (in Schow 030A)

Audrey	Weds.: 7:00-9:00
Chris	Weds.: 8:00-10:00
Spencer	Weds: 10:00-10:45
Audrey+Chris	Thurs.: 7:00-8:00
Audrey	Thurs.: 8:00-9:00
Chris	Thurs.: 9:00-10:00
Spencer+Chris	Thurs.: 10:00- 11:00

Quick Review

1. All[1] of our models of computation are described by the diagram:



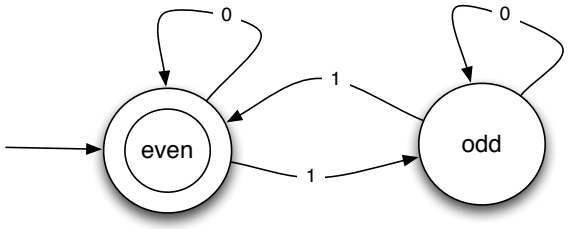
Click here to view the slides for this class

[1]This is a slight lie. We will sometimes talk about transducers which are machines that produce finite strings as their output.

- The input strings are sequences over some finite set of values called the alphabet.
  - Strings come with some fairly obvious terminology including concatenation, length, substring, suffix, and prefix.
- We call any subset of the set of all strings over an alphabet a language over that alphabet.
  - Languages come with some fairly obvious terminology including union, intersection, complement, and member of. They also come with some not so obvious terms like product[2], power, and closure.
- We will refer to the set of strings for which a given machine produces "yes" as the language recognized by that machine.

Finite State Machines

1. Last time we considered a diagram that described an algorithm/machine that could recognize the language of strings over the alphabet $\Sigma = \{0, 1\}$ that contain an even number of 1s.



2. In this diagram:
   - Circles are states; One state represents the fact that we have seen an even number of 1s so far. The other corresponds to situations where we have seen an odd number of 1s.

[2]The product of two languages is very different from the cross cross product of two sets.

- The arrows indicate when and how our state should change based on the symbols in the input.
- The circle with an arrow pointing to it is the *start* state.
- Double circles are *accept* states; The computation says "Yes" (accepts the input) if we end up in one of these states at the end of the input string. There may be 0 or more accept states.
- If we end up in one of the states that is not final, we say the computation rejects the input string.

3. An algorithm that can be described by such a diagram is called a *deterministic finite automaton* (or deterministic finite state machine).

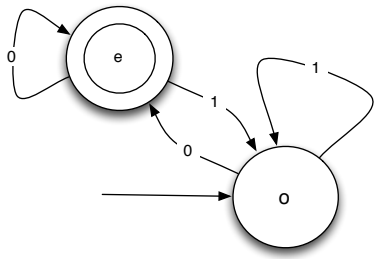
   - We will give a more formal definition of DFA shortly, but just thinking of DFAs as diagrams with states and transitions will do to get some intuition about how such machines work.

Regular Languages

1. A language is said to be *regular* if and only if it is the language of some DFA.

2. From the machine shown above from last class, we know that the set of strings of binary digits exhibiting even parity form a regular language.

3. We also discussed the machine shown below:

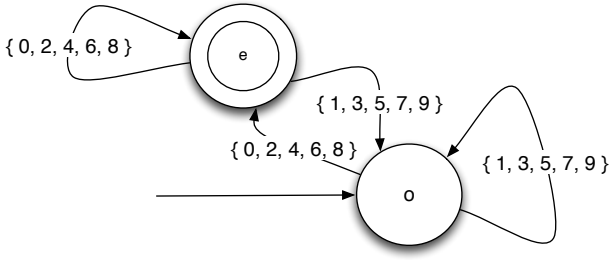


   The language of this machine is:

   (a) The set of strings of 0s and 1s, that end with a 0.

   (b) $\{w | w \in \{0,1\}^* \ \& \ w$ represents an even number in binary $\}$

4. Just to prove that we have not restricted ourself to the binary alphabet, we also discussed this machine:



   - Its alphabet is the 10 decimal digits.
   - Like our previous example, it recognizes even numbers. This time, however, the input is in decimal rather than binary.

Practice, Practice, Practice

1. To make sure you are all comfortable with the fundamentals of finite state machines before we work on making them all formal and mathematical, I would like you to work (in pairs) on describing some FSMs for the following languages. Then, I will ask you all to help me construct working solutions for everyone to see.

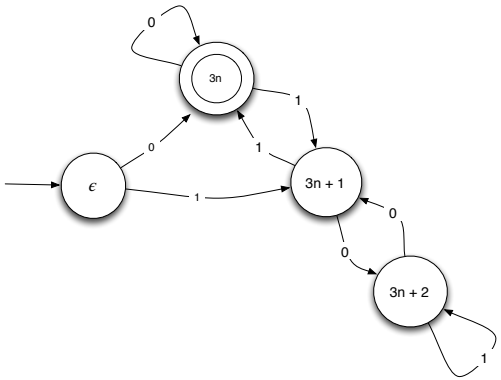
   While in class I will present all the problems first and then discuss solutions, here, I have intermixed my solutions with the problems.

2. Most FSM construction problems are totally artificial (just look at the exercises at the end of the first chapter). I have tried to think of a few examples with a bit of a practical flavor.

   - As a first exercise consider how to sketch out the state diagram for a DFA that recognizes binary sequences that represent multiples of 3.

As a hint, the machine will be a generalization of the machine we just looked at for separating odd numbers from even ones. It should have three states representing the conditions a) "The digits scanned so far form a number that is divisible by 3", b) "The digits scanned so far form a number that is one greater than a multiple of 3", and c) "The digits scanned so far form a number that is two greater than some multiple of 3".

Here is my answer to this problem:



- The state labeled "ϵ" is the start state. It ensures that the machine does not accept the empty string since it is not even a binary number.
- The transitions are designed so if the machine is in state $3n+i$ after processing some binary sequence, then the number represented by that sequence equals $3n + i$ for some n. In other words, the number represented by the binary sequence equals i mod 3.
- Numbers divisible by three equal 0 mod 3, so only the $3n$ state is final.
- The transitions between the $3n + i$ states are justified by the recognition that adding a 0 to a sequence of binary digits that represented a given number is equivalent to multiplying that number by 2 and adding a 1 is equivalent to multiplying by 2 and then adding a 1. So:
  - If the machine is in state $3n$, meaning that the digits processed so far represent the number $3n$ for some n, and the next symbol is a 0, the digits processed now represent $2 \times 3n = 3(2n) = 3n'$ if $n' = 2n$, so the machine should stay in state $3n$.
  - Similar reasoning says that since $2 \times (3n) + 1 = 3n' + 1$ if $n' = 2n$, on input 1 the machine should move from state $3n$ to state $3n + 1$ if the next symbol processed is a 1.
  - Since $2 \times (3n + 1) = 3(n2) + 2 = 3n' + 2$ if $n' = 2n$, there should be a transition on 0 from $3n + 1$ to $3n + 2$.
  - Similar reasoning explains the other transitions.

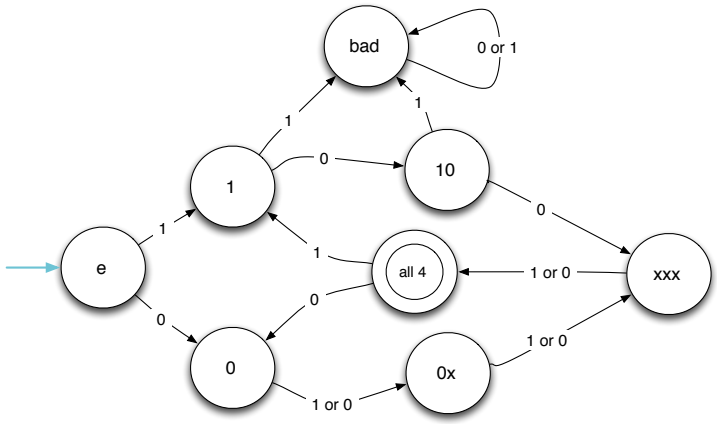
- The next example involve validity of binary strings relative to a simple scheme known as binary coded decimal (BCD for short).

  In many business applications, decimal numbers are processed, but so little arithmetic is done with them that the cost of converting to binary and then back to decimal is bigger than the processing that is actually done on the encoded numbers. In such situations, an alternative to using binary place notation is to encode each digit of a decimal number using 4 binary digits (which is enough since $2^4 > 10$) and then just string these groups of 4 together.

  For example, 361 would be represented as 001101100001 since 0011 is 3 in binary, 0110 is 6 and 0001 is 1. On the other hand 00111100001 would be invalid as a BCD encoding for two reasons: a) it breaks up as 0011 1100 001 where the last group is shorter than 4 because the total length of the sequence is not a multiple of 4 and b) the second subsequence, 1100 is 12 in binary which is bigger than any decimal digit.

  Your exercise is to build a FSM that accepts binary strings that are valid when interpreted as BCD encoded decimal numbers.

  My solution to this problem is:



This machine processes its input string four bits at a time to make sure that the input length is a multiple of four and that each substring is valid (i.e. represents a number between 0 and 9).

The roles of the states are as follows:

e The name e is short for empty. This is the start state. The machine will only be in this state when it has processed none of the input.

bad The machine enters this state and stays in this state if it finds an invalid subsequence of 4 bits.

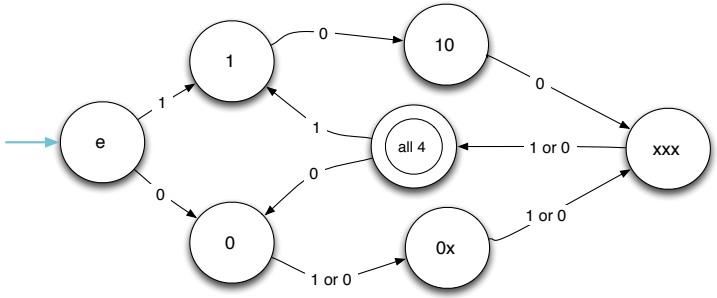
0, 1, 10 The names of these states all describe the digits of the current subsequence of 4 bits that have been scanned so far.

0x The machine enters this state after scanning two bits of a subsequence that starts with 0.

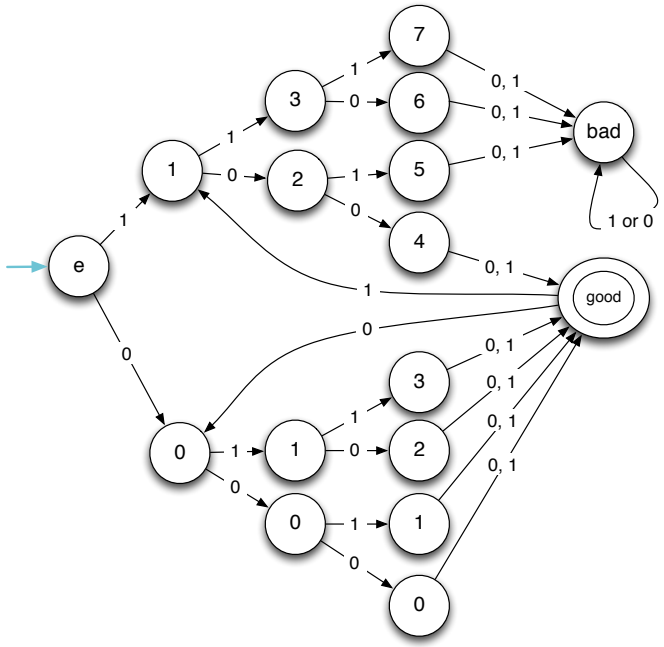
xxx The machine enters this state after scanning three bits of a valid sequence (i.e., any sequence that starts with 0 or 100).

all 4 The machine enters this state after scanning all 4 bits of a valid subsequence.

- States like "bad" are fairly common in FSMs. As a notational convention to simplify the drawing of machines, it is common to omit such states and all transitions to them from the machines diagram. In such a diagram, if the machine finds itself in a state from which there is no transition for the next input symbol, we assume it rejects the input string. The diagram below is a version of our BCD machine simplified in this way.



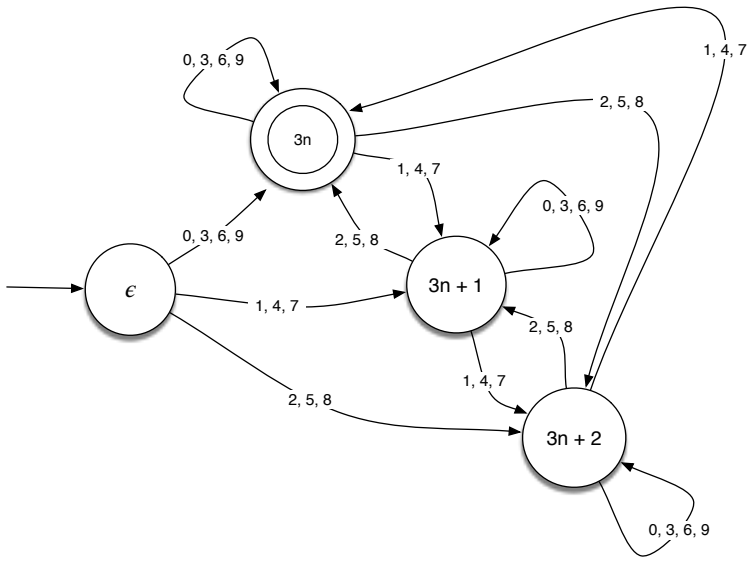
- It is worth noting that there are many ways to build a FSM for the BCD problem or any other example. Some solutions may in some sense be clearer than other. For example consider the machine:



– This machine makes it a bit clearer that it processes its input 4 symbols at a time.

– Also, rather than trying to cleverly come up with names like 0x and xxx for states, this machine lets us use simple names for most states. The labels 0 through 7 in this machine indicate the value represented by the sub-sequence of the current 4-digits being scanned.

3. Finally, since we have a bit of time to spare, we can discuss a solution to the problem of identifying numbers in base 10 that are divisible by 3. One solution is shown below.



- The simplest way to understand this machine is to remember the rule that a number is divisible by 3 if and only if the sum of its digits is divisible by 3. Thus, the state $3n+i$ should be associated with any prefix for which the sum of the digits can be written as $3n + i$ for some values of n and i.

- An alternative is to note that we can capture how we interpret a decimal number composed of a prefix of digits, p, followed by a final digit d by writing

$$value(p + d) = 10 * value(p) + d$$

Then, note that if we are interested in the value of $value(p + d)$ mod 3 we can note that this must equal $10 * value(p) + d$ mod 3 which can be rewritten as $9 * value(p) + value(p) + d$ mod 3. Since $9 * value(p)$ clearly equals 0 mod 3 we can conclude that $value(p + d)$ mod $3 = value(p) + d$ mod $3 = [(value(p)$ mod $3) + d]$ mod 3.

Formalizing DFAs

1. Now it's time to develop a mathematical formalism for deterministic finite automata. This will enable us to reason more broadly about their properties.

   Definition. A DFA is a five tuple $D = (Q, \Sigma, \delta, s, F)$ where:

   Q is a finite set of states

   Σ is the input alphabet

   $\delta : Q \times \Sigma \to Q$ is a state transition function

   $s \in Q$ is the start state

   $F \subseteq Q$ is a set of accept states

2. Using this notation, we can give a formal description of our machine that recognizes even binary numbers:

   - $Q = \{e, o\}$
   - $\Sigma = \{0, 1\}$

-

δ:	0	1
e	e	o
o	e	o

- $s = o$
- $F = \{e\}$