Illinois Institute of Technology
Department of Computer Science

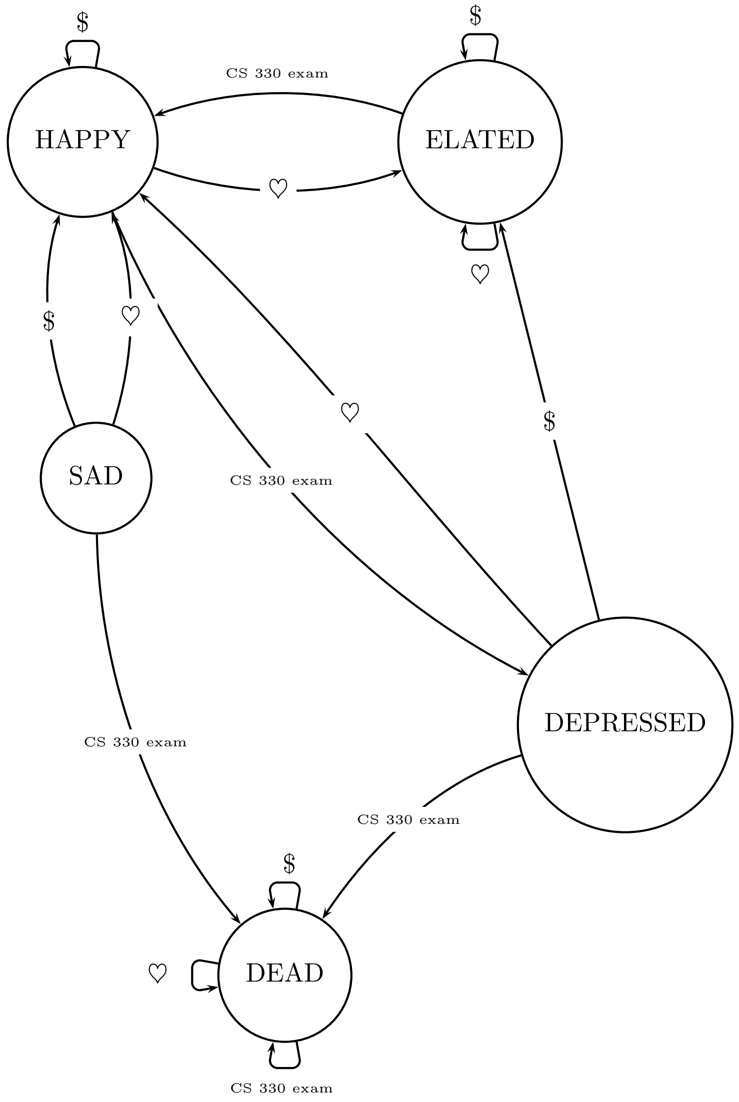
# Lecture 22: April 15, 2019

CS 330 Discrete Structures
Spring Semester, 2019

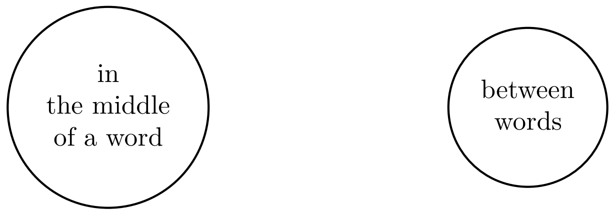
## 1 Finite state machines

### 1.1 Some examples of their use

First of all, let's take a look of a state diagram that a typical CS 330 student might follow.



One simple problem that will serve to introduce the notion of state is that of counting words: given an input string, scan the string from beginning to end and count the number of words in the string. Simply counting the number of spaces (and other word separators, such as punctuation marks) will not do, as words may be separated by more than one such separators. Rather, we need to distinguish between two *states*:

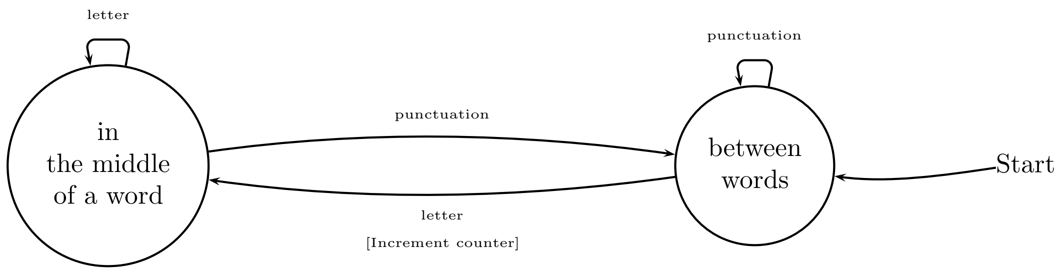


From each of these two states we can define appropriate *transitions*: that is, if we are in a state and we see an input character, what state do we emerge in? If we are in the middle of a word and we see a letter, we remain in the middle of a word. If we are between words and we see a letter, we are now in the middle of a word. Likewise, if we are in the middle of a word and we see punctuation, we are now between words, and if we are already between words and we see punctuation, we are still between words.

Where do we start? That is, before we begin processing material, which state should we be in? We should start in the "between words" state.

Finally, since we would like to count words and not just flutter between two states, when do we increment our counter? We'd like to claim that we've counted another word when we follow the transition from "between words" to "in the middle if a word."

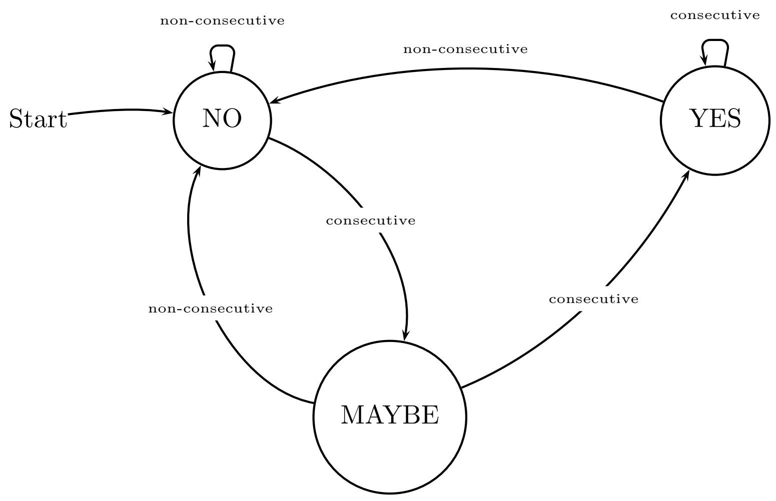
This leads to the following *state transition diagram*:



Let us consider a slightly more complicated problem. We are given a sorted list of page numbers and would like to generate an appropriate index listing — that is, runs of three or more consecutive page numbers should be collapsed. For instance, given the input 5, 6, 7, 8, 11, 12, 14, 17, 18, 19, 21, 27, 28, we would like to generate the listing 5–8, 11, 12, 14, 17–19, 21, 27, 28.

This can be modeled with three states: NO, YES and MAYBE. We are in NO when we are not in the middle of a run, in YES when we are in the middle of a run, and in MAYBE when we may be in the middle of a run.

Appropriate transitions yield the following state diagram:

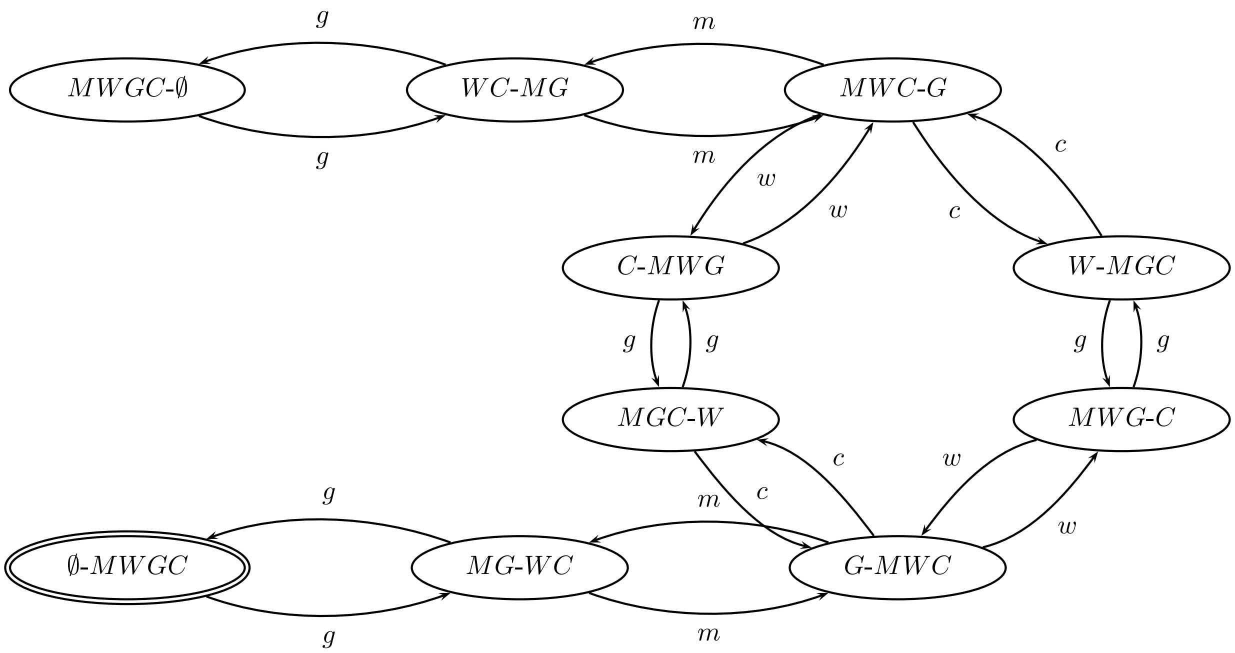


Let's consider another example. A man with a wolf, goat, and cabbage is on the left bank of a river. There is a boat large enough to carry the man and only one of the other three. The man and his entourage wish to cross to the right bank, and the man can ferry each across, one at a time. However, if the man leaves the wolf and goat unattended on either shore, the wolf will surely eat the goat. Similarly, if the goat and cabbage are left unattended, the goat will eat the cabbage. Is it possible to cross the river without the goat or cabbage being eaten?[1]

The problem is modeled by observing that the pertinent information is the occupants of each bank after a crossing. There are 16 subsets of the man ($M$), wolf ($W$), goat ($G$), and cabbage ($C$). A state corresponds to the subset that is on the left bank. States are labeled by hyphenated pairs such as $MG - WC$, where the symbols to the left of the hyphen denote the subset on the left bank; symbols to the right of the hyphen denote the subset on the right bank. Some of the 16 states, such as $GC - MW$, are fatal and may never entered by the system.

The "inputs" to the system are the actions the man takes. He may cross alone (input $m$), with the wolf (input $w$), the goat (input $g$), or cabbage (input $c$). The initial state is $MWGC$-$\emptyset$ and the final state is $\emptyset$-$MWGC$. Here is the transition diagram:

[1]This problem appears in a medieval Latin manuscript by Alcuin of York. It was used in "Gone Maggie Gone," a 2009 episode of *The Simpsons* in which Homer is trapped on one side of a river with his baby Maggie, his dog, and a bottle of poison capsules. He has only a flimsy boat so he can carry only one item at a time. If he takes the dog, Maggie might swallow some poison; if he takes the poison, the dog might bite Maggie.



There are two equally short solutions to the problem, as can be seen by searching for paths from the initial state to the final state(which is doubly circled). There are infinitely many different solutions to the problem, all but two involving useless sycles.

## 1.2 Finite state machines and juggling

A great application! See `http://en.wikipedia.org/wiki/Siteswap`

## 1.3 Finite state machines as mathematical objects

We need five entities to describe a finite state machine:

- $\Sigma$ — a finite input alphabet
- $S$ — a set of states
- $s_0 \in S$ — an initial state
- $F \subseteq S$ — a set of accept states
- $\delta : S \times \Sigma \to S$ — a transition function

Then $M = (\Sigma, S, s_0, F, \delta)$ specifies a finite state machine.

We define a *regular language* to be a language recognized by some finite state machine.[2]

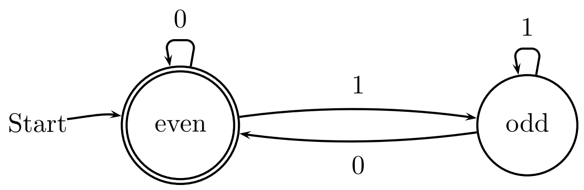
[2]Rosen follows a different treatment by defining regular languages differently and then deriving this as a theorem. This is not the approach we will be following in lecture. You have been warned.

## 1.4 Finite state machines as a computational model

As we have seen, finite state machines are a useful programming tool. They are also useful for exploring the notion of computation. A computer has a finite amount of memory, and thus can be in any of a finite (quite large, but still finite) number of states. As the computer receives input, it changes state.

Let us refine our notion of a finite state machine.[3] A finite state machine receives input symbol-by-symbol; as each symbol comes in, it changes state according to the appropriate transition. As well, certain states are declared as *accept* or *final* states. If, at the end of an input string, the machine is in an accept state, it is said to *accept* the input; otherwise it *rejects* the input. Thus a finite state machine recognizes a *language*, some subset of all finite strings over an alphabet.

Let us design a finite state machine to recognize even numbers in binary, where the input has the most significant digit first. As the machine processes the input string, it fluctuates between two states, one if the part of the input string it has seen so far is even, and the other if it is odd. The transitions are not hard to find, as a number in binary is even if it ends in a 0 and odd if it ends in a 1. Finally, we want this machine to accept if the input is even. Thus we have:



This technique can be expanded to slightly more interesting problems, for instance recognizing multiples of three or five (still in binary).

[3]Finite state machines are also known as finite automata. In particular, the variety we are developing here is a deterministic (or definite) finite automaton.