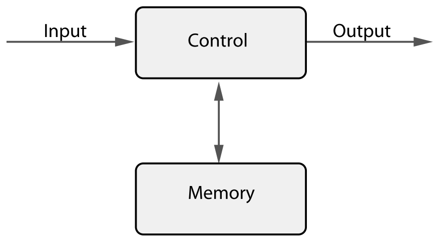
CS 361 Meeting 2 — 2/10/20

Announcements

1. Homework 1 is available on the course web page. It is due in class on Wednesday. Anonymous id numbers will be sent to you today.
2. You can use LaTeX on either the department's Linux or MacOS machines.
3. Tom's hours (this week only): Mon. 2:30-4:00, Tue. 2:00-4:00.
4. TA hours this week (this week only): Tue. 7-11 in Schow 030A.
5. Office hours schedules for the rest of the semester will be posted on course web page.

Models of Computation

1. One of the major goals this semester will be to find an example of a program we can prove it would be impossible to write.
2. To show that it is impossible to write some program, we need to precisely specify a mathematical model of what a computer is.
3. As a starting point, consider the following block diagram for a computer (which I suspect you have seen before).



4. We will actually consider three major models in this course (finite state automata, push-down automata, and Turing machines). The big difference between the three is in how the memory behaves. In particular, the form of the input and output will be the same in all three.

Click here to view the slides for this class

Strings as Input

1. Diagrams similar to that for our computer model are often used to describe mathematical functions. In that case, the input and output is often a single value from some infinite set.
2. In any real computer, the input (and output) is some sequence of symbols:
   - This might (appear to) be:
     - A sequence of characters entered on a keyboard
     - A sequence of mouse coordinates and button actions
     - A sequence of MIDI codes
     - A sequence of values produced by an accelerometer or GPS unit.
   - Ultimately, keyboards, mice, USB microphones, all convert what the user perceives as the "input" into a sequence of 0s and 1s.
3. If we modeled all input as sequences of binary digits, we would spend a lot of time figuring out how to encode things in binary. It gets messy very quickly.
4. To avoid this, we will model the input to our machines using sequences of symbols from any finite alphabet. We call such sequences *strings*.
5. With this in mind, we can define a few terms that will be important throughout the semester:

   alphabet An *alphabet* is a finite set. We typically use the symbol Σ to name our alphabet. (Occasionally, in moments of wild abandon, we might use Γ instead!) Examples include
   - $\Sigma = \{a, b, \ldots, y, z\}$ = the English alphabet with which we are all familiar.
   - $\Sigma = \{0, 1\}$ or $\{T, F\}$ = the binary alphabet.

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ = the decimal digits.

string A string is a **finite** (possibly empty) sequence of symbols from an alphabet. Examples include:

- hello world (if $\Sigma = \{a, b, \ldots, y, z, \_\}$).
- 10001 (if $\Sigma = \{0, 1\}$).
- 01267 (if $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$).

length The length of a string is just the number of symbols it contains.

ϵ The string of length 0 is called the empty string and is usually written as ϵ (rather than `""`).

concatenation Given two strings v and w, we write vw to denote the concatenation of these two strings which is just the sequence formed by placing all of the symbols in w after the symbols in v.

substring Given a string w, we say that v is a substring of w if for some strings x and y, $w = xvy$.

prefix and suffix Given a string w, if $w = xy$ we say that x is a prefix of w and y is a suffix of w.

lexicographic order is alphabetical order (or perhaps numerical order).

I am the Decider — Modeling Output

1. If we use strings to model our machine's input, it might seem to make sense to use them for output as well. If there were to be a difference between our input and output model, one might expect it to be that we would also allow infinite outputs (to account for things like a program that keeps printing digits of π forever.

2. In fact, our output model will actually be much more restrictive than our input model!
   - Our programs will all be Deciders! (Just like George Bush II.)
   - When THE DECIDER decides, the decision is not "maybe"! It is "Yes" or "No".
   - Similarly, the machines we will study will produce as a result a single "Yes" or "No" (or true or false (or 1 or 0)).

3. To understand why such a limited output model makes sense:
   (a) Remember that we want our ultimate model (the Turing Machine) to be as simple as possible as long as the simplifications themselves do not limit the things we can compute, and
   (b) Think a bit about the program to print an approximation to π that we talked about last time.
      - The version of the program we described took an input value n and printed out the first n digits of π.
      - Suppose instead we implemented a program that took an input value n and printed out just the nth digit of π.
      - Clearly, we could implement the second program iff we could implement the first.
        - Given the first program we could modify its output statements to suppress all but the last digit of output.
        - Given the second program, we could wrap a loop around its code and fool it into being run over again under the impression that its input values ranged from 1 to n.
      - Now, if when I described the second program you were thinking about decimal digits, readjust your thinking and imagine that it is a program that prints the nth bit of the binary representation of π. It is now a program that fits into our apparently very limited model, yet is arguably no less powerful or difficult to implement than a program that printed many digits of π.
      - In general, given any program that takes some input and produces arbitrary strings, we can imagine a way to represent those strings in binary and then imagine a program that takes the input to the original program and a digit position as input and tells us whether that digit of the output of the original program was 1 or 0. The new program fits our restricted model, but in some sense still provides the means to determine the complete output of the original program.

4. A program that output a boolean result can be interpreted as a program that identifies a subset of its input domain.

- If we take our binary π program and change it to output yes or no, we can interpret its behavior in an interesting way. It takes an integer as input and tells us whether that integer belongs to the set

  $$\{\ n \mid \text{the } n\text{th digit of the binary representation of } \pi \text{ is } 1\ \}$$

  That is, it "decides" whether a number is a member of a set.
- A method that describes a subset in this way is known as an *indicator* (or characteristic) function.

Based on these ideas, we will view all of the machines we describe as devices that describe sets of strings. We will say that a machine either *accepts* or *rejects* each string over its input alphabet. The set of strings accepted by a machine is called its *language*.

Languages

1. As a result, we can view the programs that can be written using our machine models as programs that describe sets of strings over an alphabet. This leads to another important definition:

   Language A language over a given alphabet is any subset of the set of all strings over that alphabet.

2. This allows us to say that a program that takes a binary encoding of a number n and says "Yes" if the n digit of π in binary is 1 and "No" if it is 0 decides the language of binary encodings of the digit positions in the binary representation of π that are ones.

3. Since languages are sets, we can certainly use standard operations and notations for talking about languages:

   membership A string is a member or element of a language if it occurs in the set which is the language.

   $\emptyset$ The empty set is a language.

   union The union of two languages is just the language containing all strings that appear in either language.

   intersection The intersection of two languages is just the language containing all strings that are members of both languages.

   complement The set of all strings that are not members of a given language. (Here, the universe is implicitly the set of all strings over the alphabet of the language.)

   power set The set of all subsets of the members of a language.

4. There are also some operations on languages that only make sense because they are sets of strings:

   Products If X and Y are languages over some alphabet, then their product, XY, is defined to be:

   $$\{xy \mid x \in X \text{ and } y \in Y\}$$

   - Note that this definition is similar to but different from the standard cross product operation. If we applied the cross product to two sets of strings we would get back a set of pairs each composed of two strings rather than a new set of strings.

   Powers If X is a language over Σ we define X^n to be the language containing only the empty string if n = 0 and XX^{n-1} otherwise.

   Closures If X is a language over Σ we define X^+, the positive closure of X, to be the union of the sets X^1, X^2, X^3, . . . and X^*, the closure of X, to be the union of X^0 and X^+.

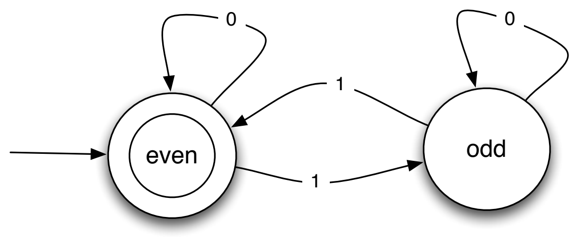
Finite State Machines

1. Let us look at a simple, but slightly practical example of a language that can be described by the kind of "decider" machine we have suggested using as our model of computation.

   - Abstractly, consider the subset of the set of strings over the alphabet $\Sigma = \{0, 1\}$ that contain an even number of 1s.
   - This abstract language has a very concrete interpretation in many computer systems.

- To detect certain hardware errors that lead to the misinterpretation of the the binary digits in a unit of data, machines often add an extra digit whose value is chosen so that the total number of 1s is even. This is called a *parity bit.*
- After the data (including the parity bit) has been stored or transmitted, the integrity of the data can be checked by ensuring that the total number of 1s is still even.

- Consider how you could handle the task of checking the parity in a large data unit as the bits arrived. Note that we could also view this as the task of *deciding* if a message had valid parity or *recognizing* the language of binary strings with valid parity.
  - You could keep a total count of the number of ones seen and at the end divide by 2.
  - It is a lot easier, however, to just remember whether the number of ones you have seen is even or odd because you would then just switch back and forth every time you saw a 1.
  - We can encode this "algorithm" with the "state transition diagram":



  - 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.
    * The arrows indicate when and how our state should change. Each arrow is labeled with one (or more) symbol from the input alphabet. We process the symbols in the input string in sequence. For each symbol, we start in some state and follow the edge from that state that is labeled with the current input symbol to the next state. This is the state in which we will begin as we process the next input symbol.
    * The circle with an arrow pointing to it is the *start* state. We position ourselves in this state as we begin processing input characters. There is only one 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.

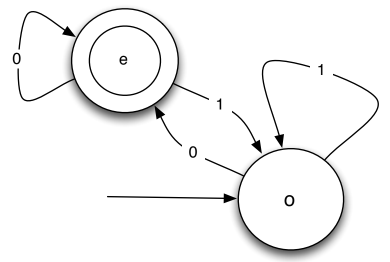
- Diagrams that describe computations in this way are called *deterministic finite automata* (or deterministic finite state machines).
  - We will give a more formal definition of DFA shortly, but just thinking of DFAs as diagrams with states and transitions will suffice to get some intuition about how 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, we know that the set of strings of binary digits exhibiting even parity form a regular language.

3. Consider the machine below:



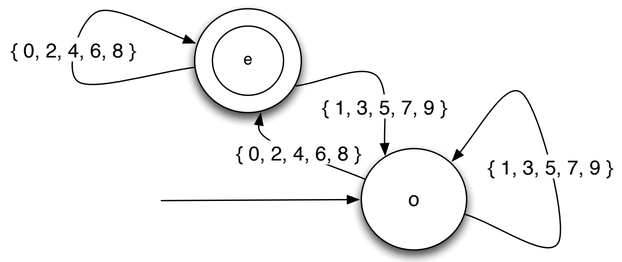
The language of this machine is:

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

(b) The set of strings of 0s and 1s, that represent an even number in binary place notation.

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

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



- Its alphabet is the 10 decimal digits.
- Take a minute to consider what language this machine recognizes.
- 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, in our next class I will ask you all to help me construct working machines for a few languages.

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".

   - The second 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.