CS 4510 Automata and Complexity Aug 24

Lecture 1

Lecturer: Frederic Faulkner *Scribe(s):*

In this class we are interested in exploring the fundamental capacities and limits of computation. But it would be very difficult to reason mathematically about your standard computer, which is filled with all kinds of circuits and gates and caches and graphics cards and... Even a simple microprocessor (like the kind you study in CS2110) has quite a few moving parts. Reasoning mathematically about such objects would be quite a challenge!

Thus computer scientists invent *models*, which are mathematical objects which can perform computations similar to those performed by traditional computers but are much simpler to describe and analyze. We can then compare the power and resource needs of these models.

In this class, we will start by analyzing a relatively "weak" model known as a Deterministic Finite Automata (DFA). DFA's are weak in the sense that they *cannot* perform every computation that a modern computer can. However, they are still valuable to study because they are present, in some form or another, in many modern devices. You might know them better by another name: *state machines*.

Suppose you are writing a program to keep track of whether some event has happened an even number of times. One possible program is the following:

$counter \leftarrow 0$

while $eventHappened()$ **do**

    $counter \leftarrow counter + 1$

end while

return $counter\%2 == 0$

This algorithm works fine, but it is rather inefficient. In particular, it requires potentially *unbounded memory*, since we are storing the total number of events but we don't know how large that will be. In fact, we do not need to store this number, as the next algorithm shows.

$isEven \leftarrow True$

while $eventHappened()$ **do**

    $isEven \leftarrow !(isEven)$

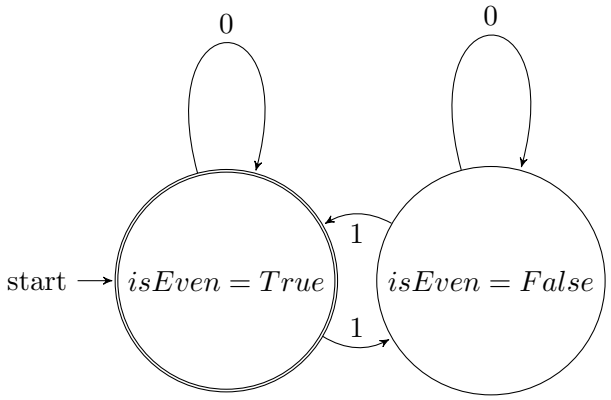
end while

return $isEven$

In contrast to the first algorithm, we only need a finite number of memory bits for this program, to store the value of $isEven$. In some sense, this program has only 2 states, since there is only one variable, $isEven$, which can only take one of two values. Let's suppose that $eventHappened()$ returns either a 1 or 0 corresponding to whether the event happened or did not, respectively. Then we could imagine making a table to track how the state of the program changes when we call $eventHappened()$.

state \ input	eventHappened=1	eventHappened=0
isEven=True	isEven=False	isEven=True
isEven=False	isEven=True	isEven=False

Similarly, we could imagine drawing this table as a diagram, where the circles represent the possible states of the program, and the arrows represent how the state changes for the given input. The double circle represents the states which cause the function to return "True".



We can do a second example. Everyone knows the joy of having to create a new password because the old one has expired. Passwords often have to meet several criteria, and sometimes the computer helpfully provides keystroke-by-keystroke updates as to the validity of our new password when we try to create one.

Let's create one of these programs. Our criteria is that any password entered must contain both a letter and a non-letter (not the strictest requirements, we know). Every time the user inputs another character, we would like to update them as to whether what they have entered so far is a valid password (green alert) or an invalid password (red alert). Here is our first attempt at such a program:

$pw \leftarrow$ ""

while $userIsTyping()$ **do**

  $pw \leftarrow pw + nextCharacter()$

  if $hasNonLetter(pw)$ & $hasLetter(pw)$ **then**

    $GreenAlert()$

  else

    $RedAlert()$

  end if

end while

Once again, this algorithm isn't quite ideal, because we need an unbounded amount of memory to store pw. (We have not put any length limits for the password, so the user might input something very long.)

However, the following algorithm accomplishes the same task without requiring unbounded memory.

$hasLetter \leftarrow False$

$hasNonLetter \leftarrow False$

while $userIsTyping()$ **do**

  if $isLetter(nextCharacter())$ **then**

    $hasLetter \leftarrow True$

  else

    $hasNonLetter \leftarrow True$

  end if

  if $hasLetter$ & $hasNonLetter$ **then**

    $GreenAlert()$

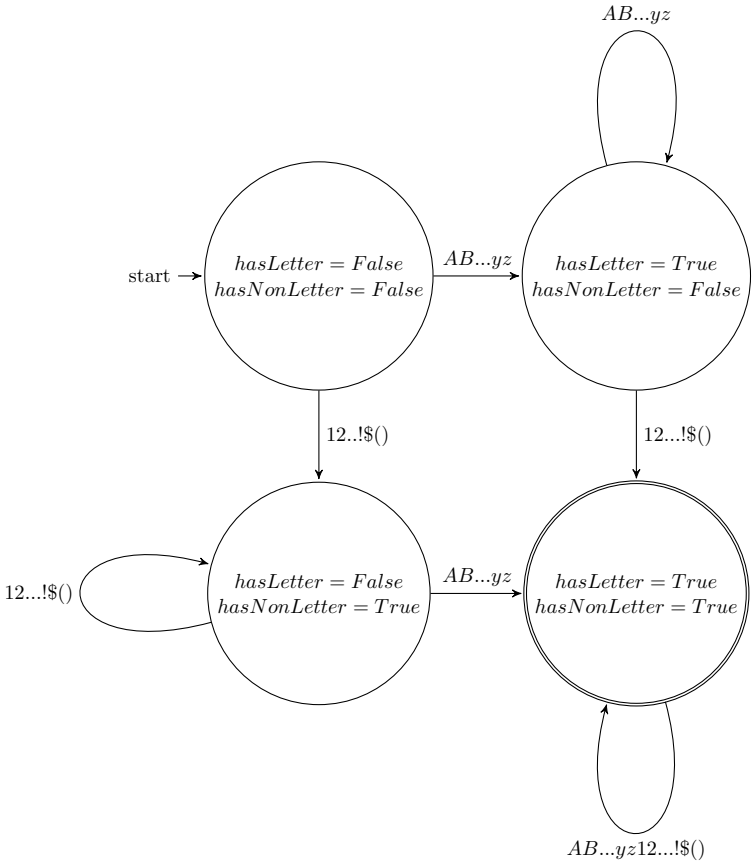
  else

    $RedAlert()$

  end if

end while

Once again, we see that as this program is run, the computer will really only be in one of four states, since there are only two variables, each of which is true or false. We can then diagram out the effect of the user's keystrokes on the internal state of the program:



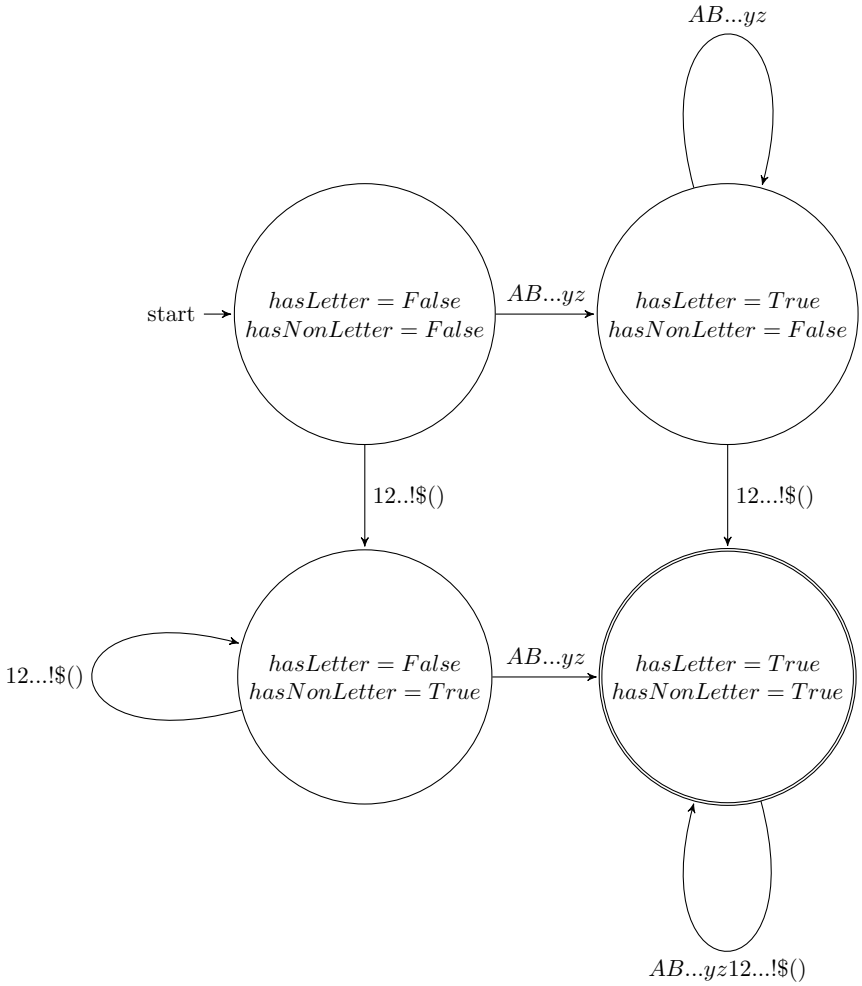
In this case, the double circle represents states which cause the Green Alert.

Now that we've seen some informal examples of DFA's, let discuss what a DFA actually is, mathematically.

Definition 1 *A DFA D is defined to be a 5-tuple $(Q, \Sigma, q_0, \delta, F)$ where*

- *Q is the set of states*
- *Σ is the set of input characters, called the alphabet*
- *$q_0 \in Q$ is the start state*
- *$F \subseteq Q$ is the set of final or accepting states*
- *$\delta : Q \times \Sigma \to Q$ is the transition function. This is what tells the machine how to move from state to state based on the current input.*

Don't let the mathematical notation scare you! We have already seen all 5 things so far today. For example let's identify the five parts of the most recent diagram, for the password checker:



- Q: this machine has four states, i.e. the four large labeled circles
- Σ: the input alphabet is whatever a person can type as part of a password, i.e. letters, numbers, and symbols.
- q_0: the machine starts in the to left state corresponding to "hasLetter=False", "hasNonLetter=False".
- F: this machine has only one accepting state, in the bottom right, represented by the doubled circle.
- δ: the transition function of the machine is represented by the arrows of the diagram. For each possible state you can be in, and each input you might see while in that state, there is an arrow to tell you what your next state will be.

Two more definitions!

Definition 2 *Formally, we say D accepts x, if there exists a sequence $q_0, ..., q_{|x|}$ of states of D such that q_0 is the initial state of D, $q_{|x|}$ is an accepting state of D, and $\delta(q_i, x[i]) = q_{i+1}$ for $i \in [0...|x| - 1]$ where δ is the transition function of D.*

D is said to reject x if it doesn't accept x.

Informally, we say a DFA D *accepts* a string x if D is in an accepting state after processing x. Thus, our password-verifier *accepts* strings which contain which contain at least one letter and one non-letter.

Definition 3 *The language of a DFA D, written $L(D)$ is the set of all strings that D accepts.*

So the *language* of our password-verifer is the set of valid passwords (according to our requirements).

Caution 1 *You do not need to explicitly list the five parts of the formal definition when you give a DFA on the homework or exam, but they do need to be implicitly present. In other words, don't forget to specify in some way, which states is the initial state or accepting states.*

Caution 2 *Your transition function must be complete, i.e. there should be an arrow for every input-state pair. There should never be a situation where you are in a state and see an input that has no arrow.*