ECS 120 Lesson 3 – Finite State Machines, Pt. 2

Oliver Kreylos

Wednesday, April 4th, 2001

1 Formal Definition of Finite State Machines

As can be seen from the first two examples of Finite State Machines, we have to specify several parts when trying to give a formal definition of one:

- How many and which states does the automaton have?
- What characters can the automaton read?
- How does the automaton move from state to state?
- Which state does the automaton start in?
- Which states are final states?

These five questions lead directly to the formal definition: A Finite State Machine M is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*.
- Σ is the *input alphabet*.
- $\delta: Q \times \Sigma \rightarrow Q$ is the *transition function*.
- $q_0 \in Q$ is the *initial state*.
- $F \subset Q$ is the set of *final states* or *accepting states*.

As an example, here is the formal definition for the pushbutton automaton shown last lecture:

$$\mathrm{PB} = \big\{\{\mathrm{On}, \mathrm{Off}\}, \{\texttt{PUSH}\}, \{(\mathrm{On}, \texttt{PUSH}) \mapsto \mathrm{Off}, (\mathrm{Off}, \texttt{PUSH}) \mapsto \mathrm{On}\}, \mathrm{Off}, \{\mathrm{On}\}\big\}$$

The transfer functions of more complicated automata are usually given in tabular form. Here is the formal definition for the doorlock automaton shown last time. For brevity, let us denote the states "idle," "3 seen," "31 seen," "314 seen," "3141 seen" and "wrong key" by q_0, q_1, q_2, q_3, q_4 and q_5, respectively:

$$\mathrm{DL} = \big\{\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}, \delta, q_0, \{q_4\}\big\}$$

where

$\delta =$

	0	1	2	3	4	5	6	7	8	9
q_0	q_5	q_5	q_5	q_1	q_5	q_5	q_5	q_5	q_5	q_5
q_1	q_5	q_2	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_2	q_5	q_5	q_5	q_5	q_3	q_5	q_5	q_5	q_5	q_5
q_3	q_5	q_4	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_4	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5
q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5

As another example, let us consider an automaton that reads an integer in binary notation (as a word over the alphabet $\{\texttt{0}, \texttt{1}\}$) and has to decide whether that integer is divisible by three. In other words, we consider an automaton D3 that accepts the language $L(\mathrm{D3}) = \big\{\, w \in \{\texttt{0}, \texttt{1}\}^* \;\big|\; \text{The integer represented by } w \text{ is divisible by } 3 \,\big\}$. Figure 1 shows the automaton as a state diagram. Here is the formal description of the automaton:

$$\mathrm{D3} = \big\{\{q_0, q_1, q_2\}, \{\texttt{0}, \texttt{1}\}, \delta, q_0, \{q_0\}\big\}$$

where

$\delta =$

	0	1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

How did we come up with this automaton? First, we have to look at the problem from an automaton's point of view. As programmers, we tend to attack the problem like this: First, store all input characters in memory. Second, convert them to an integer. Third, calculate the remainder when

dividing that integer by three. Fourth, accept the word iff the remainder is zero. Alas, this doesn't work for an automaton, for two reasons: First, an automaton cannot store its input characters (because it has no memory), and second, it cannot divide by three, because it has no arithmetic-logical unit. Thus, we have to find another way to solve the problem. A general insight about automata states that automata work by classifying input words as they are read one character at a time, and always having a partial answer ready as the input word is read. Let us first restate the problem, and then apply this insight.

The question whether an integer n is divisible by three is equivalent to the question whether its remainder when dividing by three is zero, or in formula $n \bmod 3 = 0$. Now the application of above insight says that we have to construct the final integer value of the read word as we go along, and keep track of the current remainder as we go. How is an integer value defined by a binary string if read left to right? Let us first decide that the empty word has an integer value of zero. Then, after we have read a number of binary digits, say that the integer value of the read word is v. Now we read another digit – what happens to the value? Reading another digit means that we add another digit to the right of the current binary word, effectively shifting the old digits one to the left. Shifting one to the left means multiplying the word's value by two. After shifting, the new digit is added to the word. If the new digit is zero, the word's value doesn't change; if it is one, the value increases by one. So here are the two different cases:

- Current value v, read `0`: New value is $2v$.
- Current value v, read `1`: New value is $2v + 1$.

Knowing how to keep track of the integer value of the current word, we now have to classify it according to our problem, which means we have to keep track of the current word's remainder when divided by three. There can be three different values for the remainder – 0, 1 and 2 – and we will need a state representing each of those. Now that we have decided the states, we need to create the transition function. How does the remainder change as we are reading additional digits? To find out, we need two rules from modular arithmetics:

$$(ab) \bmod n = \big((a \bmod n) \cdot (b \bmod n)\big) \bmod n$$
$$(a + b) \bmod n = \big((a \bmod n) + (b \bmod n)\big) \bmod n$$

Applying these rules to the calculation of the current value, we get

- Current remainder $v \bmod n$, read `0`: New remainder is $(2v) \bmod 3 = \big((2 \bmod 3) \cdot (v \bmod 3)\big) \bmod 3 = \big(2 \cdot (v \bmod 3)\big) \bmod 3$.
- Current remainder $v \bmod n$, read `1`: New remainder is $(2v + 1) \bmod 3 = \big(((2v) \bmod 3) + (1 \bmod 3)\big) \bmod 3 = \big((2v \bmod 3) + 1\big) \bmod 3 = \big(2(v \bmod 3) + 1\big) \bmod 3$.

Now we know how to express the new remainder in terms of the old remainder and the read character; this can directly be translated into the transition function δ by plugging in the three possible values of $v \bmod 3$. The last tasks left are to define the start state and the final states. In this case, we decided that the empty word has a value of zero and thus has a remainder of zero; therefore, the start state must be q_0. And since we only want to accept words having a remainder of zero, q_0 must also be the only final state.

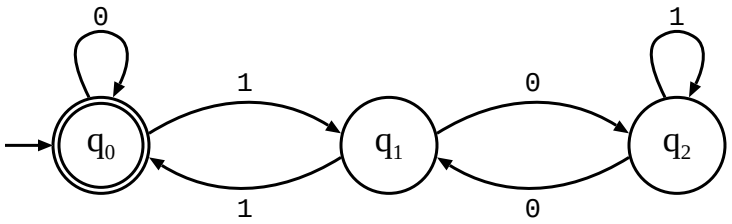

Figure 1: An automaton that decides whether an integer represented as a binary string is divisible by 3.

1.1 Discussion Problems

- Can an FSM have zero states?
- Does there have to be an outgoing arrow for each input character from each state?
- Can an FSM have zero final states? If so, what is the language accepted by an FSM with zero final states?

2 Formal Definition of Computation

Now that we can formally define an automaton, we have to formalize our description of how an automaton accepts or rejects a word. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a Finite State Machine, and let $w = w_1 w_2 \dots w_n \in \Sigma^n$ be a string of length n over M's input alphabet Σ. Then M *accepts* w if and only if there exists a sequence of states $(r_0, r_1, \dots, r_n) \in Q^n$ with the conditions

1. $r_0 = q_0$,
2. $\delta(r_{i-1}, w_i) = r_i$ for all $i = 1, 2, \dots, n$, and
3. $r_n \in F$.

These three conditions state that the automaton starts out in the start state q_0; that it follows the transition function δ from one state to the next as it reads each character w_i; and that the state the automaton ends in is a final state in F. Now we can formally define the language $L(M)$ *accepted* by automaton M as $L(M) := \left\{ w \in \Sigma^* \;\middle|\; M \text{ accepts } w \right\}$.

The above definition is taken verbatim from the textbook, but for our purposes, it is not quite formal enough. Recall that earlier we formally specified words over an alphabet using a recursive definition; to fit into our framework, the definition of computation has to match the formal definition of words. The problem in the above definition lies in the statement "let $w = w_1 w_2 \dots w_n \in \Sigma^n$ be a word of length n." Our recursive definition does not define how to write a word as a sequence of characters, so we should not use it in another formal definition[1]. To recall: Basically, we defined a word in the following way:

1. ϵ is a word.
2. If $a \in \Sigma$ is a character, and $x \in \Sigma^*$ is a word, then ax is a word.

In order to formally define computation, we have to base its definition on the definition of a word. We do that by generalizing the transfer function δ of an automaton. In its known form, δ takes a state and a single character and returns the new state the automaton will be in after reading that character.

[1] How to write a word as a sequence of characters can be deducted from the definition, but that's a different story.

We now define a generalized transition function δ^* that does not take a single character but a complete word, and returns the state the machine will be in after having read the entire word. We define $\delta^* : Q \times \Sigma^* \to Q$ as follows:

1. $\delta^*(q, \epsilon) := q$. When reading the empty word (meaning reading nothing at all), an automaton does not change its state.

2. For any $w \in \Sigma^* \setminus \{\epsilon\}$, i. e., for any non-empty word, there exist a character $a \in \Sigma$ and a word $x \in \Sigma^*$ such that $w = ax$ (inductive case of the definition of a word). We define $\delta^*(q, w) := \delta^*(\delta(q, a), x)$. When w is a non-empty word, we can split it into an initial character a and a suffix word x. When an automaton is in a state q and reads a word $w = ax$ from the left, it is going to see character a first and react to it by moving to another state. That new state q' is given by the transition function δ, and once the machine has read the first character, it will read the rest of the word, x, starting from state q'. Thus the recursive definition of δ^*: First, we apply δ to q and a, then we apply δ^* to q' the (shorter) rest of the word, x.

With the definition of δ^*, we can now define exactly what it means to say "M accepts w:" Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton, and let $w \in \Sigma^*$ be a word. Then M accepts w, iff $\delta^*(q_0, w) \in F$. In other words, a machine accepts a word if that word will take it from the start state to a final state. Similarly, we can now define the language of an automaton: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton. Then the language of the automaton M is defined as $L(M) := \left\{ w \in \Sigma^* \;\middle|\; \delta^*(q_0, w) \in F \right\}$.