

## Today's lecture: Finite Automata

- I. Finite Automata as a Model for Sequential Circuits
- II. Representing Finite Automata with Transition Graphs
- III. Formally Defining Finite Automata
- IV. Regular Languages

## Announcements

- The course newsgroup: [ubc.courses.cpsc.421](http://ubc.courses.cpsc.421). Read it. I'll post announcements to the newsgroup. You are encouraged to post questions, comments, etc. You can get extra credit by posting my mistakes – see “Bug Bounties” below.
- **Bug Bounties:** We all make mistakes. When I make a mistake, you get extra credit. Here's how it works. If I hand out a homework problem (or “Daily Question”) with an error, post a message to the newsgroup. If I agree that it's an error, you'll get extra credit. If the error is one that would prevent someone from solving the problem, you get the full value of the problem as extra credit. If it's a minor typo, you get one point. Errors of intermediate severity get intermediate extra credit. The *first* person to report the error gets the extra credit.  
  
Bug bounties are also in effect during exams. If you find an error on the midterm or final, just raise your hand and point it out to me. I'll write the correction on the white/black board and give you extra-credit on the exam. Once again, the extra credit goes to the first person to report the error.

## Reading:

**September 9:** Finite Automata – Read: *Kozen* lecture 3 or *Sipser* 1.1.

**September 12:** Regular Sets – Read: *Kozen* lecture 4 (or *Sipser* 1.1. as before).

**September 14:** Non-Determinism – Read: *Kozen* lecture 5 or *Sipser* 1.2.

**September 16:** Equivalence of DFAs and NFAs – Read: *Kozen* lecture 6 (or *Sipser* 1.2. as before).

**September 19:** Regular Expressions – Read: *Kozen* lectures 7 & 8 or *Sipser* 1.3.

**September 21:** Equivalence of Regular expressions and Finite Automata Read: *Kozen* lecture 9 (or *Sipser* 1.3. as before).

**September 23:** Nonregular Languages – Read: *Kozen* lecture 12 or *Sipser* 1.4.

**September 26:** More examples of the pumping Lemma Read: *Kozen* lecture 13 or *Sipser* 1.4.

**September 28:** Applications of finite automata

**September 30:** More applications

**October 26:** Midterm: In class.

---

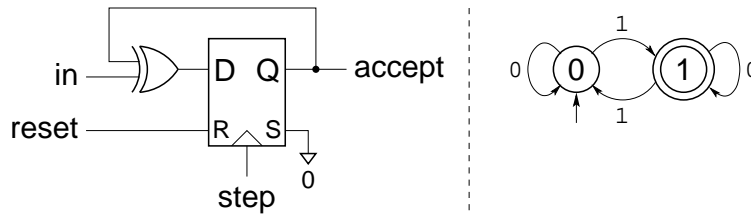


Figure 1: A Simple Finite State Machine

## I. Finite Automata as a Model for Sequential Circuits

A. The left half of figure 1 shows a simple sequential circuit.

1. When `reset` is asserted, the circuit transitions to its initial state with `accept = 0`.
2. The `step` signal is the clock input: with each rising edge of `step`, the circuit makes a transition.
3. The circuit has two states corresponding to  $Q = 0$  and  $Q = 1$ .
4. The circuit changes state if `in` is high during a rising edge of `step`.
5. Conversely, the circuit remains in its current state if `in` is low during a rising edge of `step`.

B. The right half of figure 1 shows the transition graph for this circuit.

1. Initially, the state machine is in state 0.
2. The state machine changes state when the input is a 1 and remains in its current state when its input is a 0.
3. The machine accepts an input string iff it has an odd number of ones. We can prove this by a simple induction proof. I'll skip the proof in the lecture to keep the lecture within an hour. I'll include the proof here as a simple example of how such a proof is structured. We will construct many induction proofs about automata during the semester.
4. Proof that the machine accepts a string iff it has an odd number of ones (by induction on the length of the input string):
  - a. Induction Hypothesis: The machine is in state 0 after reading an even number of ones and in state 1 after reading an odd number of ones.
  - b. Base case: the input string is empty.  
The machine starts in state 0. The empty string contains zero ones. Zero is an even number. Thus, the machine is in state 0 after reading the empty string, and this satisfies the induction hypothesis.
  - c. Induction step: assume for input  $x$ , prove for input  $x\mathbf{a}$  where  $x \in \{0, 1\}^*$ , and  $\mathbf{a} \in \{0, 1\}$ . There are four cases to consider:
    - $x$  has an even number of ones and  $\mathbf{a} = 0$ :  
After reading  $x$ , the machine is in state 0 by the induction hypothesis. Thus, after reading  $x0$ , the machine is in state 0 because the machine remains in its current state when the input is 0. Furthermore,  $x0$  has the same number of 1's as  $x$ . By the case assumption,  $x$  has an even number of 1's; thus, so does  $x0$  because they both have the same number of ones. Therefore, the induction hypothesis is maintained.
    - $x$  has an even number of ones and  $\mathbf{a} = 1$ :  
After reading  $x$ , the machine is in state 0 by the induction hypothesis. Thus, after reading  $x1$ , the machine is in state 1 because the machine changes its current state when the input is 1. Furthermore,  $x1$  has one more 1 than  $x$ . By the case assumption,  $x$  has an even number of 1's; thus,  $x1$  has an odd number because it has one more. Again, the induction hypothesis is maintained.
 The two cases when  $x$  has an odd number of ones: The arguments are similar to those above and I omit them here.

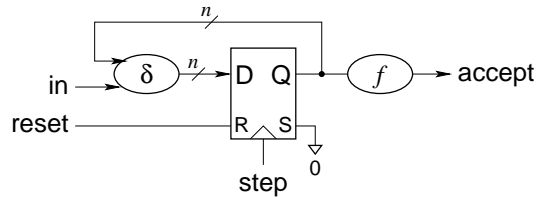






Figure 2: A Generic Sequential Decision Machine

- C. Figure 2 shows the generic implementation of a finite state machine as a sequential circuit.
1. The flip-flop from figure 1 has been replaced by  $n$  flip-flops. The circuit has  $2^n$  states.
  2. The XOR-gate from figure 1 has been replaced by the oval labeled  $\delta$ . Recall that we can implement an arbitrary boolean function using AND, OR, and NOT gates (NAND gates alone are sufficient). Thus, we can implement any next state function as a logic circuit.
  3. The circuit has  $m$  inputs. Thus, the input alphabet has  $2^m$  symbols.
  4. We can implement a finite state machine with fewer than  $2^n$  states or fewer than  $2^m$  symbols by defining  $\Sigma$  to be a subset of the  $2^m$  possible input values and  $Q$  to be a subset of the  $2^n$  flip-flop states. Then, we design the circuit for  $\delta$  such that for any  $a \in \Sigma$  and any  $q \in Q$ ,  $\delta(q, a) \in Q$ . The value of  $\delta$  for other  $a \notin \Sigma$  or  $q \notin Q$  doesn't matter – we say that such inputs are “not-allowed” and that such states are “unreachable.”

## II. Representing Finite Automata with Transition Graphs

### A. The Pieces of a Transition Graph

1. States are represented by circles
  - a. Accepting states are represented by double circles: 
  - b. Non-accepting states are represented by single circles: 
  - c. We often write the name of the state inside the circle: 

Note that the language accepted by the finite state machine is independent of the names that we choose for the states. Accordingly, we sometimes omit the names for states because the reader can make up names if s/he wants them to be named, and it won't affect the language that we are talking about.
2. Transitions are represented by directed arcs (i.e. arrows)
  - a. The direction of the arrow shows the source and destination state for the transition.
  - b. The label is a list of symbols from the input alphabet. The transition is made if the next symbol of the input string is one of the symbols listed in the label.
  - c. The labels of the outgoing arc from a state must partition the input alphabet. In other words, each symbol of the alphabet must appear in the label for exactly one outgoing arc from each state.
3. The initial state is indicated by an incoming arc with no source: 

### B. Advantages (and Disadvantages) of Transition Graphs

1. They provide visual intuition for what the finite state machine does.
2. They are simpler than circuit diagrams:
  - a. They show the behaviour of the machine rather than how that behaviour is implemented.
  - b. Thus, there can be many different ways to implement the same transition diagram. We could use different logic circuits. If we study a computing technology that doesn't use gates and flip-flops, we can still use the finite state machine model and state transition diagrams if artifacts in this new technology behave as finite state machines.

- c. Accordingly, we will use transition diagrams and the formal mathematical models for most things in this class. I will show the hardware versions occasionally, when I think that that view provides some extra intuition or a simpler explanation. When I do, I'll connect it right back to the more traditional formal automata approach very quickly. If HW makes you yawn, then you can daydream for a few minutes while I draw a schematic, and tune back in when I resume talking about sets, tuples, and graphs.
- 3. However, they are impractical for large finite state machines.
  - a. For example, a circuit with 100 flip-flops (a relatively small circuit) has over  $10^{30}$  states – it's physically impossible to draw the state transition diagram.
  - b. Likewise, the lexical analyser for a programming language such as Java may have hundreds of states. The state transition diagram is unwieldy, but it can still be helpful to understand that the lexical analyser is a collection of finite state machines.

### III. Formally Defining Finite Automata

#### A. The ingredients of a finite automaton

1. An input alphabet,  $\Sigma$ , a set of symbols.
2. A set of states,  $Q$ .
3. A transition function:  $\delta : Q \times \Sigma \rightarrow Q$ .
4. An initial state,  $q_0$ .
5. A set of accepting states:  $F \subseteq Q$ .

#### B. We can combine these to make a formal, mathematical description of a finite automaton

1. The combination is a “tuple”, that just means lump them all together.
2. The tuple is:  $(Q, \Sigma, \delta, q_0, F)$ .
3. Let  $M$  be the finite automaton  $(Q, \Sigma, \delta, q_0, F)$ . The language accepted by  $M$  is written  $L(M)$  and is defined below:
  - a. Define  $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$  recursively as shown below:

$$\begin{aligned}\hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(s \mathbf{a}) &= \delta(\hat{\delta}(q, s), \mathbf{a})\end{aligned}$$

The first line says that the machine does not change state in response to the empty string. The second line handles non-empty strings. Let  $x$  be a string of length  $n$  with  $n > 0$ . Then  $x$  can be divided into two parts,  $s$  and  $\mathbf{a}$  where  $s$  is the first  $n - 1$  characters of  $x$  and  $\mathbf{a}$  is the last character. Note that if  $x$  is of length 1, then  $s$  is empty. Now, we compute  $\hat{\delta}(q, s)$  which tells us what state the machine reaches after processing the first  $n - 1$  characters of  $x$ . Call this state  $q'$ . We then compute  $\delta(q', \mathbf{a})$  to determine the state that the machine reaches after processing the last character of  $x$ . This gives us the state that the machine reaches when starting in state  $q$  and then processing string  $x$ .

- b. String  $s$  is in language  $L(M)$  iff  $\hat{\delta}(q_0, s) \in F$ .
4. We say that this is a formal, mathematical definition because everything in the definition has a well-defined mathematical meaning: sets, functions, sequences, and tuples.

#### C. Finite automata, transition graphs, and sequential decision circuits all recognize the same set of languages.

1. Proof that finite automata and sequential decision circuits are equivalent.
  - a. Let  $C$  be a sequential decision circuit with  $m$  input wires and  $n$  flip-flops. This circuit has  $2^m$  possible input values, and  $2^n$  possible states. Thus, we define  $\Sigma = \{0 \dots 2^m - 1\}$  and  $Q = \{0 \dots 2^n - 1\}$  according to a binary interpretation of the values stored in the flip-flops. The combinational logic  $\delta$  computes a function from  $Q \times \Sigma$  to  $Q$ . This is the  $\delta$  for the finite automaton. Similarly, we define

$$F = \{q \in Q | f(q)\}$$

Finally, we define  $q_0$  such to be the state 0. Let  $M = (Q, \Sigma, \delta, q_0, F)$ . There is a straightforward proof by induction that a string  $s$  is in  $L(M)$  iff the corresponding sequence of inputs brings the sequential decision machine to a state where the **accept** output is true. The induction hypothesis is that after processing a string of length  $k$ ,  $M$  and the circuit are in corresponding states.

- b.** Let  $M = (Q, \Sigma, \delta, q_0, F)$ . We construct a sequential decision machine with  $m = \lceil \log_2 |\Sigma| \rceil$  input wires and  $n = \lceil \log_2 |Q| \rceil$  flip-flops. We map the symbols in  $\Sigma$  onto the integers  $\{0 \dots |\Sigma| - 1\}$ , and likewise map the symbols in  $Q$  onto the integers  $\{0 \dots |Q| - 1\}$ . Now, the transition function  $\delta$  maps to a function from  $\{0, 1\}^n \times \{0, 1\}^m$  to  $\{0, 1\}^n$ . We can decompose this into  $n$  functions from  $\{0, 1\}^n \times \{0, 1\}^m$  to  $\{0, 1\}$ . In the September 7 lecture, we showed that two-input AND and OR gates and inverters are sufficient to implement any boolean function. Thus, this mapped version of  $\delta$  can be implemented. Likewise, we can implement logic for  $f$  that recognizes states mapped from  $F$ . A straightforward induction proof shows that the circuit that we've just described recognizes the same language as  $M$ . The induction hypothesis again is that after processing a string of length  $k$ ,  $M$  and the circuit are in corresponding states.

2. Omitted because I need to print these notes.

#### IV. Examples of Regular Languages

- A.** A language that inputs pairs of binary numbers and accepts if the first number is greater than or equal to the second.
- B.** A simplistic spelling checker – see figure 3.
1. Requires that every word has at least one vowel or the letter 'y'.
  2. Enforces the 'i' before 'e' except after 'c' rule.
  3. Requires every 'q' to be followed by a 'u' and another vowel.
  4. Accepts many words that are not English words, such as "asdfghjkl".
  5. Rejects some perfectly fine English words, such as "cwm", "foreign", and "qat".

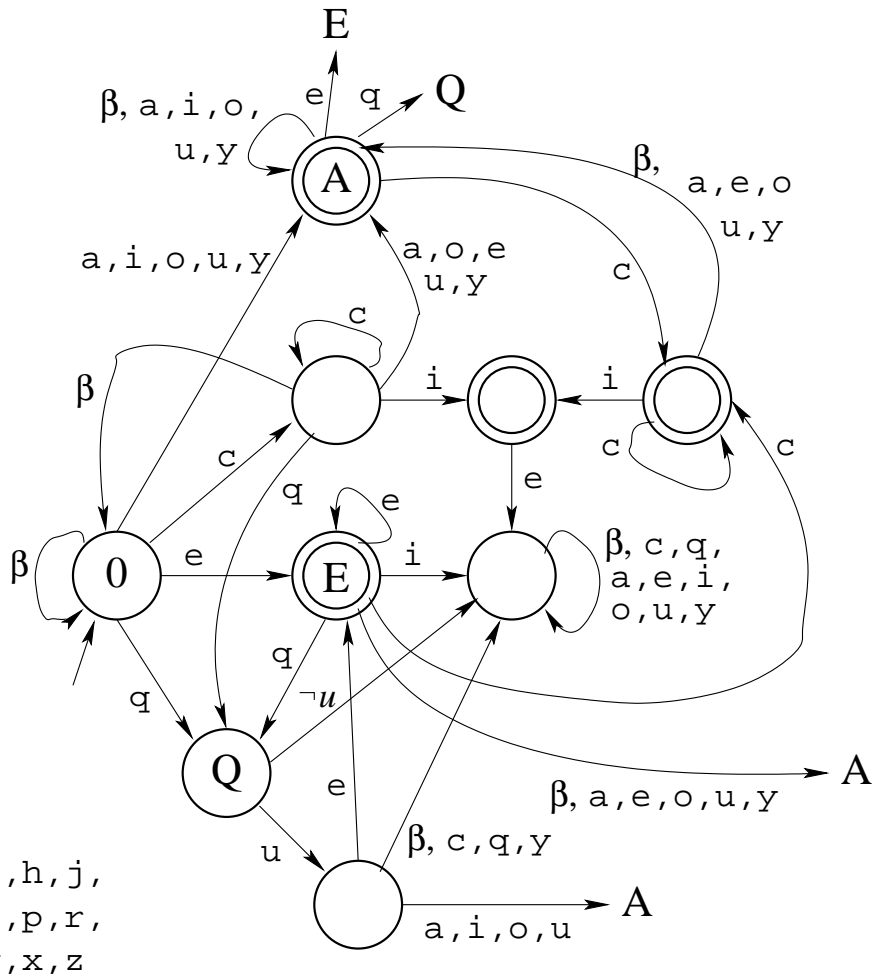


Figure 3: A Simplistic Spelling Checker