Lecture 2: Strings, Languages, DFAs

17 January 2008

This lecture covers material on strings and languages from Sipser chapter 0. Also chapter 1 up to (but not including) the formal definition of computation (i.e. pages 31–40).

1 Alphabets, strings, and languages

1.1 Alphabets

An ***alphabet*** is any *finite* set of characters.

Here are some examples for such alphabets:

(i) $\{\mathtt{0}, \mathtt{1}\}$.

(ii) $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

(iii) $\{\mathtt{0}, \mathtt{1}, \#\}$.

(iv) $\{\mathtt{a}, ...\mathtt{z}, \mathtt{A}, ...\mathtt{Z}\}$: all the letters in the English language.

(v) ASCII - this is the standard encoding schemes used by computers mappings bytes (i.e., integers in the range 0..255) to characters. As such, `a` is 65, and the space character $\textvisiblespace$ is 32.

(vi) $\{\mathtt{moveforward}, \mathtt{moveback}, \mathtt{rotate90}, \mathtt{reset}\}$.

1.2 Strings

This section should be recapping stuff already seen in discussion section 1.

A ***string*** over an alphabet Σ is a *finite* sequence of characters from Σ.

Some sample strings with alphabet (say) $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ are `abc`, `baba`, and `aaaabbbbccc`.

The ***length*** of a string x is the number of characters in x, and it is denoted by $|x|$. Thus, the length of the string $w = \mathtt{abcdef}$ is $|w| = 6$.

The ***empty string*** is denoted by ϵ, and it (of course) has length 0. The empty string is the string containing zero characters in it.

The ***concatenation*** of two strings x and w is denoted by xw, and it is the string formed by the string x followed by the string w. As a concrete example, consider $x = \mathtt{cat}$, $w = \mathtt{nip}$ and the concatenated strings $xw = \mathtt{catnip}$ and $wx = \mathtt{nipcat}$.

Naturally, concatenating with the empty string results in no change in the string. Formally, for any string x, we have that $x\epsilon = x$. As such $\epsilon\epsilon = \epsilon$.

For a string w, the string x is a ***substring*** of w if the string x appears contiguously in w.

As such, for $w =$`abcdef`
we have that `bcd` is a substring of w,
but `ace` is not a substring of w.

A string x is a ***suffix*** of w if its a substring of w appearing in the end of w. Similarly, y is a ***prefix*** of w if y is a substring of w appearing in the beginning of w.

As such, for $w =$`abcdef`
we have that `abc` is a prefix of w,
and `def` is a suffix of w.

Here is a formal definition of prefix and substring.

Definition 1.1 The string x is a ***prefix*** of a string w, if there exists a string z, such that $w = xz$.

Similarly, x is a substring of w if there exist strings y and z such that $w = yxz$.

1.3 Languages

A ***language*** is a set of strings. One special language is Σ^*, which is the set of all possible strings generated over the alphabet Σ^*. For example, if

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \quad \text{then} \quad \Sigma^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \ldots, \mathtt{aaaaaabbbaababa}, \ldots\}.$$

Namely, Σ^* is the "full" language made of characters of Σ. Naturally, any language over Σ is going to be a subset of Σ^*.

Example 1.2 The following is a language

$$L = \{\mathtt{b}, \mathtt{ba}, \mathtt{baa}, \mathtt{baaa}, \mathtt{baaaa}, ...\}.$$

Now, is the following a language?

$$\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \epsilon\}.$$

Sure – it is not a very "interesting" language because its finite, but its definitely a language.

How about $\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \emptyset\}$. Is this a language? No! Because $\emptyset$ is no a valid string (which comes to demonstrate that the empty word ϵ and $\emptyset$ are not the same creature, and they should be treated differently.

Lexicographic ordering of a set of strings is an ordering of strings that have shorter strings first, and sort the strings alphabetically within each length. Naturally, we assume that we have an order on the given alphabet.

Thus, for $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, the Lexicographic ordering of Σ^* is

$$\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \mathtt{aab}, \ldots.$$

1.3.1 Languages and set notation

Most of the time it would be more useful to use set notations to define a language; that is, define a language by the property the strings in this language posses.

For example, consider the following set of strings

$$L_1 = \left\{ x \;\middle|\; x \in \{\mathtt{a}, \mathtt{b}\}^* \text{ and } |x| \text{ is even} \right\}.$$

In words, L_1 is the language of all strings made out of $\mathtt{a}, \mathtt{b}$ that have even length.

Next, consider the following set

$$L_2 = \left\{ x \;\middle|\; \text{there is a } w \text{ such that } xw = \mathtt{illinois} \right\}.$$

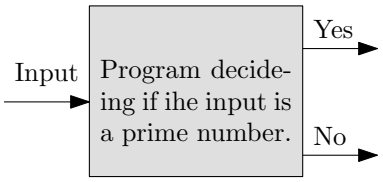
So L_2 is the language made out of all prefixes of L_2. We can write L_2 explicitly, but its tedious. Indeed,

$$L_2 = \{\epsilon, \mathtt{i}, \mathtt{il}, \mathtt{ill}, \mathtt{illi}, \mathtt{illin}, \mathtt{illino}, \mathtt{illinoi}, \mathtt{illinois}\}.$$

1.3.2 Why should we care about languages?

Consider the language L_{primes} that contains all strings over $\Sigma = \{0, 1, \ldots, 9\}$ which are prime numbers. If we can build a fast computer program (or an automata) that can tell us whether a string s (i.e., a number) is in L_{primes}, then we decide if a number is prime or not. And this is a very useful program to have, since most encryption schemes currently used by computers (i.e., RSA) rely on the ability to find very large prime numbers.

Let us state it explicitly: The ability to decide if a word is in a specific language (like L_{primes}) is equivalent to performing a computational task (which might be extremely non-trivial). You can think about this schematically, as a program that gets as input a number (i.e., string made out of digits), and decides if it is prime or not. If the input is a prime number, it outputs `Yes` and otherwise it outputs `No`. See figure on the right.



1.4 Strings and programs

An text file (i.e., source code of a program) is a long one dimensional string with special ⟨NL⟩ (i.e., newline) characters that instruct the computer how to display the file on the screen. That is, the special ⟨NL⟩ characters instruct the computer to start a new line. Thus, the text file

```
if x=y then
  jump up and down and scream.
```

Is in fact encoded on the computer as the string

`if␣x=y␣then`⟨NL⟩`␣␣jump␣up␣and␣down␣and␣scream.`

Here, ␣ denote the special space character and ⟨NL⟩ is the new-line character.

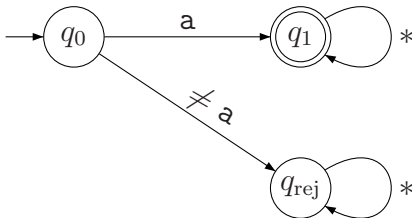
It would be sometime useful to use similar "complicated" encoding schemes, with subparts separated by # or $ rather than by ⟨NL⟩.

Program input and output can be consider to be files. So a standard program can be taught of as a function that maps strings to strings.[1] That is $P : \Sigma^* \rightarrow \Sigma^*$. Most machines in this class map input strings to two outputs: "yes" and "no". A few automatas and most real-world machines produce more complex output.

2 State machines

2.1 A simple automata

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all strings starting with a.



Here $*$ represents any possible character.

Notice key pieces of this machine: three states, q_0 is the start state (arrow coming in), q_1 is the final state (double circle), transition arcs.

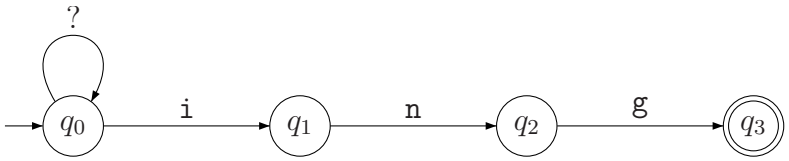
To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{a}, \texttt{aa}, \texttt{ab}, \texttt{aaa}, \ldots\}$.

2.2 Another automata

(This section is optional and can be skipped in the lecture.)

Here is a simple ***state machine*** (i.e., finite automaton) M that accepts all ASCII strings ending with ing.



Notice key pieces of this machine: four states, q_0 is the start state (arrow coming in), q_3 is the final state (double circle), transition arcs.

To run the machine, we start at the start state. On each input character, we follow the corresponding arc. When we run out of input characters, we answer "yes" or "no", depending on whether we are in the final state.

[1] Here, we are considering simple programs that just read some input, and print out output, without fancy windows and stuff like that.

The language of a machine M is the set of strings it accepts, written $L(M)$. In this case $L(M) = \{\texttt{walking}, \texttt{flying}, \texttt{ing}, \ldots\}$.

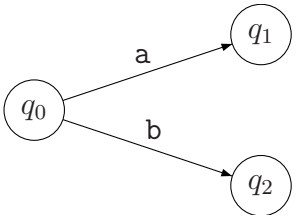
2.3 What automatas are good for?

People use the technology of automatas in real-world applications:

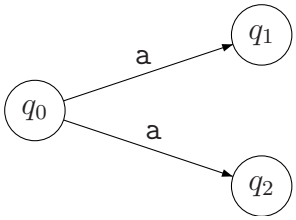
- Find all files containing `-ing` (grep)
- Translate each `-ing` into `-iG` (finite-state transducer)
- How often do words in Chomsky's latest book end in `-ing`?

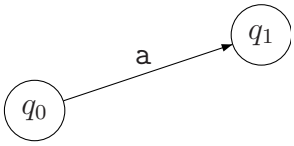
2.4 DFA - deterministic finite automata

We will start by studying ***deterministic finite automata*** (DFA). Each node in a deterministic machine has exactly one outgoing transition for each character in the alphabet. That is, if the alphabet is $\{\texttt{a}, \texttt{b}\}$, then all nodes need to look like

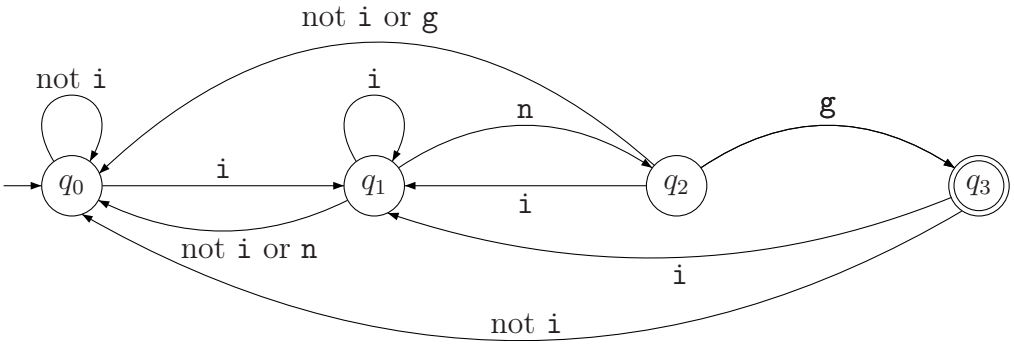


Both of the following are bad, where $q_1 \neq q_2$ and the right hand machine has no outgoing transition for the input character `b`.





So our `-ing` detector would be redrawn as:

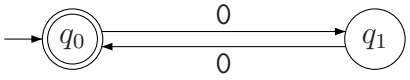


3 More examples of DFAs

3.1 Number of characters is even

Input: $\Sigma = \{0\}$.

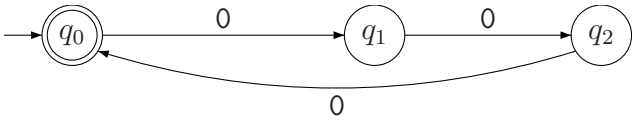
Accept: all strings in which the number of characters is even.



3.2 Number of characters is divisible by 3

Input: $\Sigma = \{0\}$.

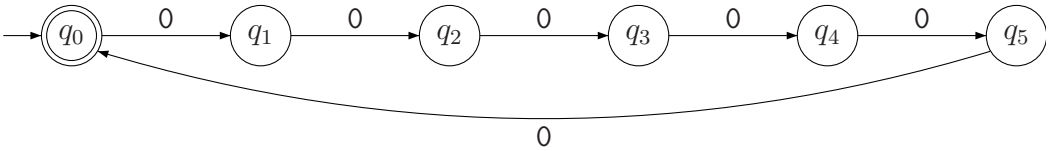
Accept: all strings in which the number of characters is divisible by 3.



3.3 Number of characters is divisible by 6

Input: $\Sigma = \{0\}$.

Accept: all strings in which the number of characters is divisible by 6.

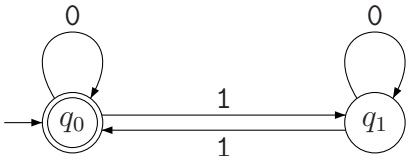


This example is especially interesting, because we can achieve the same purpose, by observing that $n \bmod 6 = 0$ if and only if $n \bmod 2 = 0$ and $n \bmod 3 = 0$ (i.e., to be divisible by 6, a number has to be divisible by 2 and divisible by 3 [a generalization of this idea is known as the Chinese remainder theorem]). So, we could run the two automatas of Section 3.1 and Section 3.2 in parallel (replicating each input character to each one of the two automatas), and accept only if both automatas are in an accept state. This idea would become more useful later in the course, as it provide a building operation to construct complicated automatas from simple automatas.

3.4 Number of ones is even

Input is a string over $\Sigma = \{0, 1\}$.

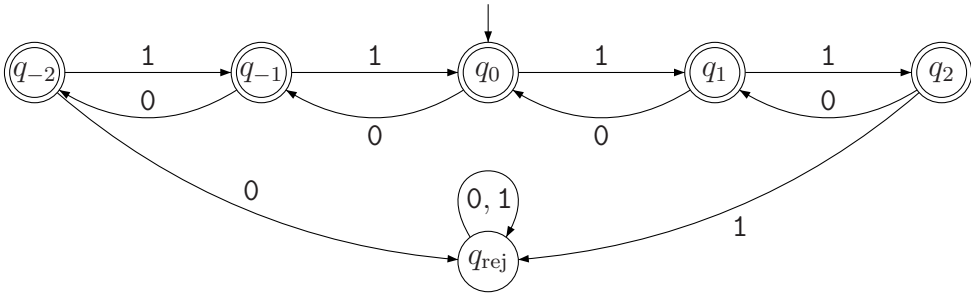
Accept: all strings in which the number of ones is even.



3.5 Number of zero and ones is always within two of each other

Input is a string over $\Sigma = \{0, 1\}$.

Accept: all strings in which the difference between the number of ones and zeros in any prefix of the string is in the range $-2, \ldots, 2$. For example, the language contains ϵ, 0, 001, and 1101. You even have an extended sequence of one character e.g. 001111, but it depends what preceded it. So 111100 isn't in the language.



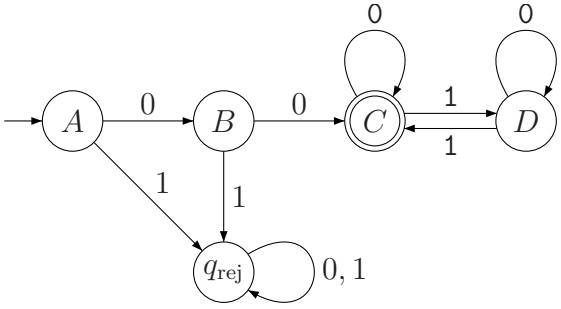
Notice that the names of the states reflect their role in the computation. When you come to analyze these machines formally, good names for states often makes your life much easier. BTW, the language of this DFA is

$$L(M) = \left\{ w \;\middle|\; w \in \{0,1\}^* \text{ and for every } x \text{ that is a prefix of } w, |\#1(x) - \#0(x)| \leq 2 \right\}.$$

3.6 More complex language

The input is strings over $\Sigma = \{0, 1\}$.

Accept: all strings of the form $00w$, where w contains an even number of ones.



You can name states anything you want. Names of the form q_X are often convenient, because they remind you of what's a state. And people often make the initial state q_0. But this isn't obligatory.

4 The pieces of a DFA

To specify a DFA (***deterministic finite automata***), we need to describe

– a (finite) alphabet

– a (finite) set of states

– which state is the start state?

– which states are the final states?

– what is the transition from each state, on each input character?