Contents

1

Lectures

Change

This is page 2
Printer: Opaq

Lecture 1

Course Roadmap and Historical Perspective

The goal of this course is to understand the foundations of computation. We will ask some very basic questions, such as:

- What does it mean for a function to be computable?
- Are there any noncomputable functions?
- How does computational power depend on programming constructs?

These questions may appear simple, but they are not. They have intrigued scientists for decades, and the subject is still far from closed.

In the quest for answers to these questions, we will encounter along the way some fundamental and pervasive concepts: *state, transition, nondeterminism, reduction*, and *undecidability*, to name a few. Some of the most important achievements in theoretical computer science have been the crystallization of these concepts. They have shown a remarkable persistence, even as technology changes from day to day. They are crucial for every good computer scientist to know, so that they can be recognized when they are encountered, as they surely will be.

Various models of computation have been proposed over the years, all of which capture some fundamental aspect of computation. We will concentrate on the following three classes of models, in order of increasing power:

(i) (finite memory) finite automata, regular expressions

(ii) (finite memory + stack) pushdown automata

(iii) (unrestricted)

- Turing machines (Alan Turing [120])
- Post systems (Emil Post [99, 100])
- μ-recursive functions (Kurt Gödel [51], Jacques Herbrand)
- λ-calculus (Alonzo Church [23], Stephen C. Kleene [66])
- combinatory logic (Moses Schönfinkel 1924 [111], Haskell B. Curry [29])

These systems were developed long before computers existed. Nowadays one could add PASCAL, FORTRAN, BASIC, LISP, SCHEME, C++, JAVA, or any sufficiently powerful programming language to this list.

In parallel with and independent of the development of these models of computation, the linguist Noam Chomsky was attempting to formalize the notion of *grammar* and *language*. This effort resulted in the definition of the *Chomsky Hierarchy*, a hierarchy of language classes defined by grammars of increasing complexity:

(i) right-linear grammars and languages

(ii) context-free grammars and languages

(iii) unrestricted grammars and languages

Although grammars and machine models appear quite different on a superficial level, the process of parsing a sentence in a language bears a strong resemblance to computation. Upon closer inspection, it turns out that each of the grammar types (i–iii) are equivalent in computational power to the machine models (i–iii) above, respectively, in a certain well-defined sense. There is even a fourth natural class called the *context-sensitive* grammars and languages, which fits in between (ii) and (iii) above and which corresponds to a certain natural class of machine models called *linear bounded automata.*

It is quite surprising that a naturally defined hierarchy in one field should correspond so closely to a naturally defined hierarchy in a completely different field. Could this be mere coincidence?

Abstraction

The machine models mentioned above were first identified in the same way that theories in physics or any other scientific discipline arise. When studying real-world phenomena, one becomes aware of recurring patterns and themes that appear in various guises. These guises may differ substantially on a superficial level, but may bear enough resemblance to one another to suggest that there are common underlying principles at work. When this happens, it makes sense to try to construct an abstract model that captures these underlying principles in the simplest possible way, devoid of the unimportant details of each particular manifestation. This is the process of *abstraction*. Abstraction is the essence of scientific progress, because it focusses attention on the important principles, unencumbered by irrelevant details.

Perhaps the most striking example of this phenomenon we will see is the formalization of the concept of *effective computability*. This quest started around the beginning of the 20th century with the development of the *formalist* school of mathematics, championed by the philosopher Bertrand Russell and the mathematician David Hilbert. They wanted to reduce all of mathematics to the formal manipulation of symbols.

Of course, the formal manipulation of symbols is a form of computation, although there were no computers around at the time. However, there certainly existed an awareness of computation and algorithms. Mathematicians, logicians, and philosophers knew a constructive method when they saw it. There followed several attempts to come to grips with the general notion of *effective computability*. Several definitions emerged (Turing machines, Post systems, *etc.*), each with its own peculiarities, and differing radically in appearance. However, it turned out that as different as all these formalisms appeared to be, they could all simulate one another, thus they were all computationally equivalent.

The formalist program was eventually shattered by Kurt Gödel's Incompleteness Theorem, which states that no matter how strong a deductive system for number theory you take, it will always be possible to construct simple statements that are true but unprovable. This theorem is widely regarded as one of the crowning intellectual achievements of the twentieth century. It is essentially a statement about computability, and we will be in a position to give a full account of it by the end of the course.

The process of abstraction is inherently mathematical. It involves building models that capture observed behavior in the simplest possible way. Although we will consider plenty of concrete examples and applications of these models, we will work primarily mathematically, in terms of their properties. We will always be as explicit as possible about these properties. We will usually start with definitions, then subsequently reason purely in terms of those definitions. For some, this will undoubtedly be a new way of thinking, but it is a skill that is worth cultivating.

Keep in mind that a large intellectual effort often goes into coming up with just the right definition or model that captures the essence of the principle at hand with the least amount of extraneous baggage. After the fact, the reader often sees only the finished product, and is not exposed to all the misguided false attempts and pitfalls that were encountered along the way. Remember that it took many years of intellectual struggle to arrive at the theory as it exists today. This is not to say that the book is closed—far from it!

Lecture 2

Strings and Sets

Decision Problems vs. Functions

A *decision problem* is a function with a one-bit output: “yes” or “no”. To specify a decision problem, one must specify

- the set A of possible inputs
- the subset $B \subseteq A$ of “yes” instances.

For example, to decide if a given graph is connected, the set of possible inputs is the set of all (encodings of) graphs, and the “yes” instances are the connected graphs. To decide if a given number is a prime, the set of possible inputs is the set of all (binary encodings of) integers, and the “yes” instances are the primes.

In this course we will mostly consider decision problems as opposed to functions with more general outputs. We do this for mathematical simplicity, and because the behavior we want to study is already present at this level.

Strings

Now to our first abstraction: we will always take the set of possible inputs to a decision problem to be the set of finite-length strings over some fixed finite alphabet (formal definitions below). We do this for uniformity and simplicity. Other types of data—graphs, the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, trees, even programs—can be encoded naturally as strings. By making this abstraction, we only have to deal with one data type and a few basic operations.

Definition 2.1

- An *alphabet* is any finite set. For example, we might use the alphabet $\{0, 1, 2, \ldots, 9\}$ if we are talking about decimal numbers; the set of all ASCII characters if talking about text; $\{0, 1\}$ if talking about bit strings. The only restriction is that the alphabet be finite.

  When speaking about an arbitrary finite alphabet abstractly, we usually denote it by the Greek letter Σ. We call elements of Σ *letters* or *symbols* and denote them by $a, b, c, \ldots$

  Often we do not care at all about the nature of the elements of Σ, only that there are finitely many of them.

- A *string* over Σ is any finite-length sequence of elements of Σ. Example: if $\Sigma = \{a, b\}$, then $aabab$ is a string over Σ of length five. We use $x, y, z, \ldots$ to refer to strings.

- The *length* of a string x is the number of symbols in x. The length of x is denoted $|x|$. For example, $|aabab| = 5$.

- There is a unique string of length 0 over Σ called the *null string* or *empty string* and denoted by ϵ (Greek epsilon, not to be confused with the symbol for set containment $\in$). Thus $|\epsilon| = 0$.

- We write a^n for a string of a's of length n. For example, $a^5 = aaaaa$, $a^1 = a$, and $a^0 = \epsilon$. Formally, a^n is defined inductively:

$$\begin{aligned} a^0 &\overset{\text{def}}{=} \epsilon \\ a^{n+1} &\overset{\text{def}}{=} a^n a \ . \end{aligned}$$

- The set of all strings over alphabet Σ is denoted Σ^*. For example,

$$\begin{aligned} \{a, b\}^* &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \\ \{a\}^* &= \{\epsilon, a, aa, aaa, aaaa, \ldots\} \\ &= \{a^n \mid n \geq 0\} \ . \end{aligned}$$

□

By convention, we take

$$\varnothing^* \stackrel{\text{def}}{=} \{\epsilon\} ,$$

where $\varnothing$ denotes the empty set. This may seem a bit strange, but there is good mathematical justification for it, which will become apparent shortly.

If Σ is nonempty, then Σ^* is an infinite set of finite-length strings. Be careful not to confuse strings and sets. We won't be seeing any infinite strings until much later in the course. Here are some differences between strings and sets:

- $\{a, b\} = \{b, a\}$ but $ab \neq ba$
- $\{a, a, b\} = \{a, b\}$ but $aab \neq ab$.

Note also that $\varnothing$, $\{\epsilon\}$, and ϵ are three different things. The first is a set with no elements; the second is a set with one element, namely ϵ; and the last is a string, not a set.

Operations on Strings

The operation of *concatenation* takes two strings x and y and makes a new string xy by putting them together end to end. The string xy is called the *concatenation* of x and y. Note that xy and yx are different in general. Some useful properties of concatenation are:

- concatenation is *associative*: $(xy)z = x(yz)$
- the null string ϵ is an *identity* for concatenation: $\epsilon x = x\epsilon = x$
- $|xy| = |x| + |y|$

A special case of the last equation is $a^m a^n = a^{m+n}$ for all $m, n \geq 0$.

A *monoid* is any algebraic structure consisting of a set with an associative binary operation and an identity for that operation. By our definitions above, the set Σ^* with string concatenation as the binary operation and ϵ as the identity is a monoid. We will be seeing some other examples later on in the course.

Definition 2.2

- We write x^n for the string obtained by concatenating n copies of x. For example, $(aab)^5 = aabaabaabaabaab$, $(aab)^1 = aab$, and $(aab)^0 = \epsilon$. Formally, x^n is defined inductively:

$$\begin{aligned} x^0 &\stackrel{\text{def}}{=} \epsilon \\ x^{n+1} &\stackrel{\text{def}}{=} x^n x \ . \end{aligned}$$

- If $a \in \Sigma$ and $x \in \Sigma^*$, we write $\#a(x)$ for the number of a's in x. For example, $\#0(001101001000) = 8$ and $\#1(00000) = 0$.

$\square$

Operations on Sets

We usually denote sets of strings (subsets of Σ^*) by $A, B, C, \ldots$ The *cardinality* (number of elements) of set A is denoted $|A|$. The empty set $\varnothing$ is the unique set of cardinality 0.

Let's define some useful operations on sets. Some of these you have probably seen before, some probably not.

- *Set union*:

  $$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ or } x \in B\} .$$

  In other words, x is in the union of A and B iff[1] either x is in A or x is in B. For example, $\{a, ab\} \cup \{ab, aab\} = \{a, ab, aab\}$.

- *Set intersection*:

  $$A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ and } x \in B\} .$$

  In other words, x is in the intersection of A and B iff x is in both A and B. For example, $\{a, ab\} \cap \{ab, aab\} = \{ab\}$.

- *Complement in* Σ^*:

  $$\sim A \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid x \notin A\} .$$

  For example,

  $$\sim\{\text{strings in } \Sigma^* \text{ of even length}\} = \{\text{strings in } \Sigma^* \text{ of odd length}\} .$$

  Unlike $\cup$ and $\cap$, the definition of $\sim$ depends on Σ^*. The set $\sim A$ is sometimes denoted $\Sigma^* - A$ to emphasize this dependence.

- *Set concatenation*:

  $$AB \stackrel{\text{def}}{=} \{xy \mid x \in A \text{ and } y \in B\}$$

  In other words, z is in AB iff z can be written as a concatenation of two strings x and y, where $x \in A$ and $y \in B$. For example, $\{a, ab\}\{b, ba\} = \{ab, aba, abb, abba\}$. When forming a set concatenation, you include *all* strings that can be obtained in this way. Note that AB and BA are different sets in general. For example, $\{b, ba\}\{a, ab\} = \{ba, bab, baa, baab\}$.

[1]iff = if and only if

- The *powers* A^n of a set A are defined inductively as follows:

$$\begin{aligned} A^0 &\stackrel{\text{def}}{=} \{\epsilon\} \\ A^{n+1} &\stackrel{\text{def}}{=} AA^n . \end{aligned}$$

  In other words, A^n is formed by concatenating n copies of A together. Taking $A^0 = \{\epsilon\}$ makes the property $A^{m+n} = A^m A^n$ hold, even when one of m or n is 0. For example,

$$\begin{aligned} \{ab, aab\}^0 &= \{\epsilon\} \\ \{ab, aab\}^1 &= \{ab, aab\} \\ \{ab, aab\}^2 &= \{abab, abaab, aabab, aabaab\} \\ \{ab, aab\}^3 &= \{ababab, ababaab, abaabab, aababab, \\ &\qquad abaabaab, aababaab, aabaabab, aabaabaab\} . \end{aligned}$$

  Also,

$$\begin{aligned} \{a, b\}^n &= \{x \in \{a, b\}^* \mid |x| = n\} \\ &= \{\text{strings over } \{a, b\} \text{ of length } n\} . \end{aligned}$$

- The *asterate* A^* of a set A is the union of all finite powers of A:

$$\begin{aligned} A^* &\stackrel{\text{def}}{=} \bigcup_{n\geq 0} A^n \\ &= A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots \end{aligned}$$

  Another way to say this is

$$A^* = \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\} .$$

  Note n can be 0; thus the null string ϵ is in A^* for any A.

  We previously defined Σ^* to be the set of all finite-length strings over the alphabet Σ. This is exactly the asterate of the set Σ, so our notation is consistent.

- We define A^+ to be the union of all *nonzero* powers of A:

$$A^+ \stackrel{\text{def}}{=} AA^* = \bigcup_{n\geq 1} A^n .$$

Here are some useful properties of these set operations:

- Set union, set intersection, and set concatenation are *associative*:

$$\begin{aligned} (A \cup B) \cup C &= A \cup (B \cup C) \\ (A \cap B) \cap C &= A \cap (B \cap C) \\ (AB)C &= A(BC) . \end{aligned}$$

- Set union and intersection are *commutative*:
$$A \cup B = B \cup A$$
$$A \cap B = B \cap A \ .$$
As noted above, set concatenation is not.

- The null set $\varnothing$ is an *identity* for $\cup$:
$$A \cup \varnothing = \varnothing \cup A = A \ .$$

- The set $\{\epsilon\}$ is an identity for set concatenation:
$$\{\epsilon\}A = A\{\epsilon\} = A \ .$$

- The null set $\varnothing$ is an *annihilator* for set concatenation:
$$A\varnothing = \varnothing A = \varnothing \ .$$

- Set union and intersection *distribute* over each other:
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \ .$$

- Set concatenation distributes over union:
$$A(B \cup C) = AB \cup AC$$
$$(A \cup B)C = AC \cup BC \ .$$
In fact, concatenation distributes over the union of any family of sets. If $\{B_i \mid i \in I\}$ is any family of sets indexed by another set I, finite or infinite, then
$$A(\bigcup_{i \in I} B_i) = \bigcup_{i \in I} AB_i$$
$$(\bigcup_{i \in I} B_i)A = \bigcup_{i \in I} B_iA \ .$$
Here $\bigcup_{i \in I} B_i$ denotes the union of all the sets B_i for $i \in I$. An element x is in this union iff it is in one of the B_i.

  Set concatenation does *not* distribute over intersection. For example, take $A = \{a, ab\}$, $B = \{b\}$, $C = \{\epsilon\}$, and see what you get when you compute $A(B \cap C)$ and $AB \cap AC$.

- The *De Morgan Laws* hold:
$$\sim\!(A \cup B) = \sim\! A \cap \sim\! B$$
$$\sim\!(A \cap B) = \sim\! A \cup \sim\! B \ .$$

- The asterate operation * satisfies the following properties:

$$\begin{aligned} A^*A^* &= A^* \\ A^{**} &= A^* \\ A^* &= \{\epsilon\} \cup AA^* = \{\epsilon\} \cup A^*A \\ \varnothing^* &= \{\epsilon\}\ . \end{aligned}$$

Lecture 3

Finite Automata and Regular Sets

States and Transitions

Intuitively, a *state* of a system is an instantaneous description of that system, a snapshot of reality frozen in time. A state gives all relevant information necessary to determine how the system can evolve from that point on. *Transitions* are changes of state; they can happen spontaneously or in response to external inputs.

We assume that state transitions are instantaneous. This is a mathematical abstraction. In reality, transitions usually take time. Clock cycles in digital computers enforce this abstraction and allow us to treat computers as digital instead of analog devices.

There are innumerable examples of state transition systems in the real world: electronic circuits, digital watches, elevators, Rubik's cube ($54!/9!^6$ states and twelve transitions, not counting peeling the little sticky squares off), the game of Life (2^k states on a screen with k cells, one transition).

A system that consists of only finitely many states and transitions among them is called a *finite-state transition system.* We model these abstractly by a mathematical model called a *finite automaton.*

Finite Automata

Formally, a *deterministic finite automaton* (DFA) is a structure

$$M = (Q,\ \Sigma,\ \delta,\ s,\ F)$$

where:

- Q is a finite set; elements of Q are called *states*
- Σ is a finite set, the *input alphabet*
- $\delta : Q \times \Sigma \to Q$ is the *transition function* (recall $Q \times \Sigma$ is the set of ordered pairs $\{(q, a) \mid q \in Q \text{ and } a \in \Sigma\}$). Intuitively, δ is a function that tells which state to move to in response to an input: if M is in state q and sees input a, it moves to state $\delta(q, a)$.
- $s \in Q$ is the *start state*
- F is a subset of Q; elements of F are called *accept* or *final states.*

When you specify a finite automaton, you must give all five parts. Automata may be specified in this set theoretic form, or as a transition diagram or table as in the example below.

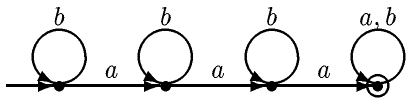
Example 3.1 Here is an example of a simple four-state finite automaton. We'll take the set of states to be $\{0, 1, 2, 3\}$, the input alphabet to be $\{a, b\}$, the start state to be 0, the set of accept states to be $\{3\}$, and the transition function to be

$$\begin{aligned}
&\delta(0, a) = 1 \\
&\delta(1, a) = 2 \\
&\delta(2, a) = \delta(3, a) = 3 \\
&\delta(q, b) = q\ , \quad q \in \{0, 1, 2, 3\}\ .
\end{aligned}$$

All parts of the automaton are completely specified. We can also specify the automaton by means of a table

		a	b
$\rightarrow$	0	1	0
	1	2	1
	2	3	2
	$3F$	3	3

or transition diagram



The final states are indicated by an F in the table and by a circle in the transition diagram. In both, the start state is indicated by $\rightarrow$. The states in the transition diagram from left to right correspond to the states $0, 1, 2, 3$ in the table. One advantage of transition diagrams is that you don't have to name the states. □

Another convenient representation of finite automata is transition matrices; see Miscellaneous Exercise 7.

Informally, here is how a finite automaton operates. An input can be any string $x \in \Sigma^*$. Put a pebble down on the start state s. Scan the input string x from left to right, one symbol at a time, moving the pebble according to δ: if the next symbol of x is b and the pebble is on state q, move the pebble to $\delta(q, b)$. When we come to the end of the input string, the pebble is on some state p. The string x is said to be *accepted* by the machine M if $p \in F$, *rejected* if $p \notin F$. There is no formal mechanism for scanning or moving the pebble; these are just intuitive devices.

For example, the automaton of Example 3.1, starting in its start state 0, will be in state 3 after scanning the input string *baabbaab*, so that string is accepted; whereas it will be in state 2 after scanning the string *babbbab*, so that string is rejected. For this automaton, a moment's thought reveals that when scanning any input string, the automaton will be in state 0 if it has seen no a's, 1 if it has seen one a, 2 if it has seen two a's, and 3 if it has seen three or more a's.

This is how we do formally what we just described informally above. We first define a function

$$\widehat{\delta} : Q \times \Sigma^* \rightarrow Q$$

from δ by induction on the length of x:

$$\widehat{\delta}(q, \epsilon) \overset{\text{def}}{=} q \tag{3.1}$$

$$\widehat{\delta}(q, xa) \overset{\text{def}}{=} \delta(\widehat{\delta}(q, x), a) \tag{3.2}$$

The function $\widehat{\delta}$ maps a state q and a string x to a new state $\widehat{\delta}(q, x)$. Intuitively, $\widehat{\delta}$ is the multi-step version of δ. The state $\widehat{\delta}(q, x)$ is the state M ends up in when started in state q and fed the input x, moving in response to each symbol of x according to δ. Equation (3.1) is the basis of the inductive definition; it says that the machine doesn't move anywhere under the null input. Equation (3.2) is the induction step; it says that the state reachable from q under input string xa is the state reachable from p under input symbol a, where p is the state reachable from q under input string x.

Note that the second argument to $\widehat{\delta}$ can be any string in Σ^*, not just a string of length one as with δ; but $\widehat{\delta}$ and δ agree on strings of length one:

$$\begin{aligned} \widehat{\delta}(q,a) &= \widehat{\delta}(q,\epsilon a) && \text{since } a = \epsilon a \\ &= \delta(\widehat{\delta}(q,\epsilon),a) && \text{by (3.2), taking } x = \epsilon \\ &= \delta(q,a) && \text{by (3.1).} \end{aligned}$$

Formally, a string x is said to be *accepted* by the automaton M if

$$\widehat{\delta}(s,x) \in F$$

and *rejected* by the automaton M if

$$\widehat{\delta}(s,x) \notin F$$

where s is the start state and F is the set of accept states. This captures formally the intuitive notion of acceptance and rejection described above.

The *set* or *language accepted by* M is the set of all strings accepted by M, and is denoted $L(M)$:

$$L(M) \overset{\text{def}}{=} \{x \in \Sigma^* \mid \widehat{\delta}(s,x) \in F\} .$$

A subset $A \subseteq \Sigma^*$ is said to be *regular* if $A = L(M)$ for some finite automaton M. The set of strings accepted by the automaton of Example 3.1 is the set

$$\{x \in \{a,b\}^* \mid x \text{ contains at least three } a\text{'s}\} ,$$

so this is a regular set.

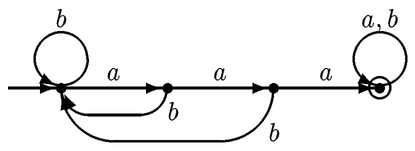
Here is another example of a regular set and a finite automaton accepting it.

Example 3.2 Consider the set

$$\begin{aligned} &\{xaaay \mid x,y \in \{a,b\}^*\} \\ &\quad = \{x \in \{a,b\}^* \mid x \text{ contains a substring of three consecutive } a\text{'s}\} . \end{aligned}$$

For example, *baabaaaab* is in the set and should be accepted, whereas *babbabab* is not in the set and should be rejected (because the three a's are not consecutive). Here is an automaton for this set, specified in both table and transition diagram form:

		a	b
$\rightarrow$	0	1	0
	1	2	0
	2	3	0
	$3F$	3	3



□

The idea here is that you use the states to count the number of consecutive a's you have seen. If you haven't seen three a's in a row and you see a b, you must go back to the start. Once you have seen three a's in a row, though, you stay in the accept state.

Lecture 4

More on Regular Sets

Here is another example of a regular set that is a little little harder than the example given last time. Consider the set

$$\{x \in \{0,1\}^* \mid x \text{ represents a multiple of three in binary}\} \tag{4.1}$$

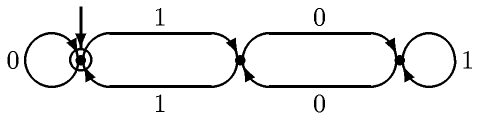
(leading zeros permitted, ϵ represents the number 0). For example, the following binary strings represent multiples of three and should be accepted:

binary	*decimal equivalent*
0	0
11	3
110	6
1001	9
1100	12
1111	15
10010	18
$\vdots$	$\vdots$

Strings not representing multiples of three should be rejected. Here is an automaton accepting the set (4.1):

		0	1
$\rightarrow$	$\mathbf{0}F$	$\mathbf{0}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{0}$
	$\mathbf{2}$	$\mathbf{1}$	$\mathbf{2}$

The states $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$ are written in boldface to distinguish them from the input symbols $0, 1$.



In the diagram, the states are $\mathbf{0}$, $\mathbf{1}$, $\mathbf{2}$ from left to right. We prove that this automaton accepts exactly the set (4.1) by induction on the length of the input string. First we associate a meaning to each state[2]:

if the number represented by the string scanned so far is	*then the machine will be in state*
0 mod 3	$\mathbf{0}$
1 mod 3	$\mathbf{1}$
2 mod 3	$\mathbf{2}$

Let $\#x$ denote the number represented by string x in binary. For example,

$$\begin{aligned}
\#\epsilon &= 0\\
\#0 &= 0\\
\#11 &= 3\\
\#100 &= 4\ ,
\end{aligned}$$

etc. Formally, we want to show for any string x in $\{0,1\}^*$,

$$\begin{aligned}
\widehat{\delta}(\mathbf{0},x) &= \mathbf{0} \text{ iff } \#x \equiv 0 \bmod 3 \qquad (4.2)\\
\widehat{\delta}(\mathbf{0},x) &= \mathbf{1} \text{ iff } \#x \equiv 1 \bmod 3\\
\widehat{\delta}(\mathbf{0},x) &= \mathbf{2} \text{ iff } \#x \equiv 2 \bmod 3
\end{aligned}$$

or in short,

$$\widehat{\delta}(\mathbf{0},x) = \#x \bmod 3\ . \qquad (4.3)$$

[2]Here $a \bmod n$ denotes the remainder when dividing a by n using ordinary integer division. We also write $a \equiv b \bmod n$ (read: a is congruent to b modulo n) to mean that a and b have the same remainder when divided by n; *i.e.*, that n divides $b - a$.

This will be our induction hypothesis. The final result we want, namely (4.2), is a weaker consequence of (4.3), but we need the more general statement (4.3) for the induction hypothesis.

We have by elementary number theory that

$$\begin{aligned}\#(x0) &= 2(\#x) + 0\\ \#(x1) &= 2(\#x) + 1\end{aligned}$$

or in short,

$$\#(xc) = 2(\#x) + c \tag{4.4}$$

for $c \in \{0, 1\}$. From the machine above, we see that for any state $q \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ and input symbol $c \in \{0, 1\}$,

$$\delta(q, c) = (2q + c) \bmod 3\ . \tag{4.5}$$

This can be verified by checking all six cases corresponding to possible choices of q and c. (In fact, (4.5) would have been a great way to *define* the transition function formally—then we wouldn't have had to prove it!) Now we use the inductive definition of $\widehat{\delta}$ to show (4.3) by induction on $|x|$.

Basis

For $|x| = 0$, *i.e.* $x = \epsilon$,

$$\begin{aligned}\widehat{\delta}(\mathbf{0}, \epsilon) &= \mathbf{0} && \text{by definition of } \widehat{\delta}\\ &= \#\epsilon && \text{since } \#\epsilon = 0\\ &= \#\epsilon \bmod 3\ .\end{aligned}$$

Induction step

Assuming (4.3) is true for $x \in \{0, 1\}^*$, we show it is true for xc, where $c \in \{0, 1\}$.

$$\begin{aligned}\widehat{\delta}(\mathbf{0}, xc) &= \delta(\widehat{\delta}(\mathbf{0}, x), c) && \text{definition of } \widehat{\delta}\\ &= \delta(\#x \bmod 3, c) && \text{induction hypothesis}\\ &= (2(\#x \bmod 3) + c) \bmod 3 && \text{by (4.5)}\\ &= (2(\#x) + c) \bmod 3 && \text{elementary number theory}\\ &= \#xc \bmod 3 && \text{by (4.4).}\end{aligned}$$

Some Closure Properties of Regular Sets

For $A, B \subseteq \Sigma^*$, recall the definitions

$$\begin{array}{lll}
A \cup B & = \{x \mid x \in A \text{ or } x \in B\} & \text{union} \\
A \cap B & = \{x \mid x \in A \text{ and } x \in B\} & \text{intersection} \\
\sim A & = \{x \in \Sigma^* \mid x \notin A\} & \text{complement} \\
AB & = \{xy \mid x \in A \text{ and } y \in B\} & \text{concatenation} \\
A^* & = \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\} & \\
 & = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots & \text{asterate}
\end{array}$$

Don't confuse set concatenation with string concatenation. Sometimes $\sim A$ is written $\Sigma^* - A$.

We show below that if A and B are regular, then so are $A \cup B$, $A \cap B$, and $\sim A$. We'll show later that AB and A^* are also regular.

The Product Construction

Assume that A and B are regular. Then there are automata

$$\begin{aligned}
M_1 &= (Q_1,\ \Sigma,\ \delta_1,\ s_1,\ F_1) \\
M_2 &= (Q_2,\ \Sigma,\ \delta_2,\ s_2,\ F_2)
\end{aligned}$$

with $L(M_1) = A$ and $L(M_2) = B$. To show that $A \cap B$ is regular, we will build an automaton M_3 such that $L(M_3) = A \cap B$.

Intuitively, M_3 will have the states of M_1 and M_2 encoded somehow in its states. On input $x \in \Sigma^*$, it will simulate M_1 and M_2 simultaneously on x, accepting iff both M_1 and M_2 would accept. Think about putting a pebble down on the start state of M_1 and another on the start state of M_2. As the input symbols come in, move both pebbles according to the rules of each machine. Accept if both pebbles occupy accept states in their respective machines when the end of the input string is reached.

Formally, let

$$M_3 = (Q_3,\ \Sigma,\ \delta_3,\ s_3,\ F_3)$$

where

$$\begin{aligned}
Q_3 &= Q_1 \times Q_2 = \{(p, q) \mid p \in Q_1 \text{ and } q \in Q_2\} \\
F_3 &= F_1 \times F_2 = \{(p, q) \mid p \in F_1 \text{ and } q \in F_2\} \\
s_3 &= (s_1, s_2)
\end{aligned}$$

and let

$$\delta_3 : Q_3 \times \Sigma \to Q_3$$

be the transition function defined by

$$\delta_3((p,q),a) = (\delta_1(p,a), \delta_2(q,a)) \ .$$

The automaton M_3 is called the *product* of M_1 and M_2. A state (p,q) of M_3 encodes a configuration of pebbles on M_1 and M_2.

Recall the inductive definition (3.1), (3.2) of the extended transition function $\widehat{\delta}$ from Lecture 2. Applied to δ_3, this gives:

$$\begin{aligned} \widehat{\delta}_3((p,q),\epsilon) &= (p,q) \\ \widehat{\delta}_3((p,q),xa) &= \delta_3(\widehat{\delta}_3((p,q),x),a) \ . \end{aligned}$$

Lemma 4.1 *For all $x \in \Sigma^*$,*

$$\widehat{\delta}_3((p,q),x) = (\widehat{\delta}_1(p,x), \widehat{\delta}_2(q,x)) \ .$$

Proof. By induction on $|x|$.

Basis

For $|x| = 0$, *i.e.* $x = \epsilon$,

$$\widehat{\delta}_3((p,q),\epsilon) = (p,q) = (\widehat{\delta}_1(p,\epsilon), \widehat{\delta}_2(q,\epsilon)) \ .$$

Induction step

Assuming the Lemma holds for $x \in \Sigma^*$, we show it holds for xa, where $a \in \Sigma$.

$$\begin{aligned} \widehat{\delta}_3((p,q),xa) &= \delta_3(\widehat{\delta}_3((p,q),x),a) && \text{definition of } \widehat{\delta}_3 \\ &= \delta_3((\widehat{\delta}_1(p,x), \widehat{\delta}_2(q,x)),a) && \text{induction hypothesis} \\ &= (\delta_1(\widehat{\delta}_1(p,x),a), \delta_2(\widehat{\delta}_2(q,x)),a)) && \text{definition of } \delta_3 \\ &= (\widehat{\delta}_1(p,xa), \widehat{\delta}_2(q,xa)) && \text{definition of } \widehat{\delta}_1 \text{ and } \widehat{\delta}_2. \end{aligned}$$

□

Theorem 4.2 $L(M_3) = L(M_1) \cap L(M_2)$.

Proof. For all $x \in \Sigma^*$,

$$\begin{array}{ll}
x \in L(M_3) & \\
\iff \widehat{\delta}_3(s_3, x) \in F_3 & \text{definition of acceptance} \\
\iff \widehat{\delta}_3((s_1, s_2), x) \in F_1 \times F_2 & \text{definition of } s_3 \text{ and } F_3 \\
\iff (\widehat{\delta}_1(s_1, x), \widehat{\delta}_2(s_2, x)) \in F_1 \times F_2 & \text{Lemma 4.1} \\
\iff \widehat{\delta}_1(s_1, x) \in F_1 \text{ and } \widehat{\delta}_2(s_2, x) \in F_2 & \text{definition of set product} \\
\iff x \in L(M_1) \text{ and } x \in L(M_2) & \text{definition of acceptance} \\
\iff x \in L(M_1) \cap L(M_2) & \text{definition of intersection.}
\end{array}$$

□

To show that regular sets are closed under complement, take an automaton accepting A and interchange the set of accept and nonaccept states. The resulting automaton accepts exactly when the original automaton would reject, so the set accepted is $\sim A$.

Once we know regular sets are closed under $\cap$ and $\sim$, it follows that they are closed under $\cup$ by one of the De Morgan Laws:

$$A \cup B = \sim(\sim A \cap \sim B)\ .$$

If you use the constructions for $\cap$ and $\sim$ given above, this gives an automaton for $A \cup B$ which looks exactly like the product automaton for $A \cap B$, except that the accept states are

$$F_3 = \{(p, q) \mid p \in F_1 \text{ or } q \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$$

instead of $F_1 \times F_2$.

Historical Notes

Finite state transition systems were introduced by McCulloch and Pitts in 1943 [84]. Deterministic finite automata in the form presented here were studied by Kleene [70]. Our notation is borrowed from Hopcroft and Ullman [60].

Lecture 5

Nondeterministic Finite Automata

Nondeterminism

Nondeterminism is an important abstraction in computer science. It refers to situations in which the next state of a computation is not uniquely determined by the current state. Nondeterminism arises in real life when there is incomplete information about the state or when there are external forces at work that can affect the course of a computation. For example, the behavior of a process in a distributed system might depend on messages from other processes that arrive at unpredictable times with unpredictable contents.

Nondeterminism is also important in the design of efficient algorithms. There are many instances of important combinatorial problems with efficient nondeterministic solutions, but no known efficient deterministic solution. The famous $P = NP$ problem—whether all problems solvable in nondeterministic polynomial time can be solved in deterministic polynomial time—is a major open problem in computer science, and arguably one of the most important open problems in all of mathematics.

In nondeterministic situations, we may not know how a computation will evolve, but we may have some idea of the range of possibilities. This is modeled in automata theory by allowing automata to have multiple-valued transition functions. For decision problems, the dominant paradigm is *guess and verify*—on a given input, guess a successful computation or proof that the input is a "yes" instance of the decision problem, and verify that the guess is indeed correct.

In this lecture and the next, we will show how nondeterminism is incorporated naturally in the context of finite automata. One might think that adding nondeterminism might increase expressive power, but in fact for finite automata it does not: in terms of the sets accepted, nondeterministic finite automata are no more powerful than deterministic ones. In other words, for every nondeterministic finite automaton, there is a deterministic one accepting the same set. However, nondeterministic machines may be exponentially more succinct.

Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) is one for which the next state is not necessarily uniquely determined by the current state and input symbol. In a deterministic automaton, there is exactly one start state and exactly one transition out of each state for each symbol in Σ, given by the function δ. In a nondeterministic automaton, there may be one, more than one, or zero. The set of *possible* next states that the automaton may move to from a particular state q in response to a particular input symbol a is part of the specification of the automaton, but there is no mechanism for deciding which one will actually be taken. Formally, we won't be able to represent this with a function $\delta : Q \times \Sigma \to Q$ anymore; we will have to use something more general. Also, a nondeterministic automaton does not have a unique start state, but may have many, and may start in any one of them.

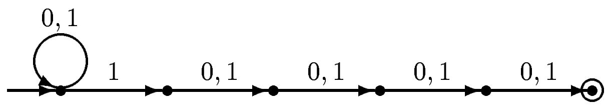
Informally, a nondeterministic automaton is said to *accept* its input x if it is possible to start in some start state and scan x, moving according to the transition rules (making choices along the way if there are more than one possible next state) such that when the end of x is reached, the machine is in an accept state. Because the start state is not determined and because of the choices along the way, there may be several possible paths through the automaton in response to the input x; some may lead to accept states while others may lead to reject states. The automaton is said to *accept* x if *at least one* computation path on input x starting from *at least one* start state leads to an accept state. The automaton is said to *reject* x if *no* computation path on input x from *any* start state leads to an accept state. Again, there is no mechanism for determining which state to start in or which of the possible next moves to take in response to an input symbol.

For example, consider the set

$$A = \{x \in \{0,1\}^* \mid \text{the fifth symbol from the right is } 1\} .$$

Thus $11010010 \in A$ but $11000010 \notin A$.

Here is a six-state nondeterministic automaton accepting A:



There is only one start state, namely the leftmost, and only one accept state, namely the rightmost. The automaton is not deterministic because there are two transitions from the leftmost state labeled 1 (one back to itself and one to the second state) and no transitions from the rightmost state. This automaton accepts the set A, because for any string x whose fifth symbol from the right is 1, *there exists* a sequence of legal transitions leading from the start state to the accept state (it moves from the first state to the second when it scans the fifth symbol from the right); and for any string x whose fifth symbol from the right is 0, there is *no possible* sequence of legal transitions leading to the accept state, no matter what choices it makes (recall that to accept, the machine must be in an accept state when the end of the input string is reached).

Informally, we can think of the machine as *guessing* when it sees a 1 whether to take the transition from the first state to the second, *i.e.* whether that 1 is fifth from the right. But it is not enough to guess, the machine must also *verify* that its guess was correct; this is the purpose of the tail of the automaton leading to the accept state.

To show formally that this machine accepts the set A, we would have to argue that for any string $x \in A$, *i.e.* for any string with a 1 fifth from the right, there is a lucky sequence of guesses that leads to an accept state when the end of x is reached; but for any string $x \notin A$, *i.e.* for any string with a 0 fifth from the right, *no* sequence of guesses leads to an accept state when the end of x is reached, no matter how lucky the automaton is.

There does exist a deterministic automaton accepting the set A, but any such automaton must have at least $2^5 = 32$ states, since a deterministic machine essentially has to remember the last five symbols seen.

Equivalence of DFAs and NFAs

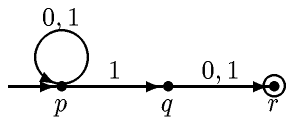
We will prove a rather remarkable fact: in terms of the sets accepted, nondeterministic finite automata are no more powerful than deterministic ones. In other words, for every nondeterministic finite automaton, there is a deterministic one accepting the same set. The deterministic automaton, however, may require more states.

This theorem can be proved using the so-called *subset construction.* Here is the idea. Given a nondeterministic machine N, think of putting pebbles on the states to keep track of all the states N could possibly be in after scanning an initial substring of the input. We start with pebbles on all the start states of the nondeterministic machine. Say after scanning some initial substring x of the input string, we have pebbles on some set P of states, and say P is the set of all states N could possibly be in after scanning x, depending on the nondeterministic choices that N could have made so far. If input symbol b comes in, pick the pebbles up off the states of P and put a pebble down on each state reachable from a state in P under input symbol b. Let P' be the new set of states covered by pebbles. Then P' is the set of states that N could possibly be in after scanning xb.

Although for a state q of N, there may be many possible next states after scanning b, note that the set P' is uniquely determined by P and b. We will thus build a deterministic automaton M *whose states are these sets.* That is, a state of M will be a *set* of states of N. These will be the P, P', *etc.* The start state of M will be the *set* of start states of N, indicating that we start with one pebble on each of the start states of N. A final state of M will be any set P containing a final state of N, since we want to accept x if it is possible for N to have made choices while scanning x leading to an accept state of N.

It takes a stretch of the imagination to regard a set of states of N as a single state of M. Let's illustrate the construction with a shortened version of the example above. Consider the set

$$A = \{x \in \{0,1\}^* \mid \text{the second symbol from the right is } 1\} .$$



Label the states p, q, r from left to right, as illustrated. The states of M will be *subsets* of the set of states of N. In this example there are eight such subsets:

$$\varnothing,\ \{p\},\ \{q\},\ \{r\},\ \{p,q\},\ \{p,r\},\ \{q,r\},\ \{p,q,r\} .$$

Here is the deterministic automaton M:

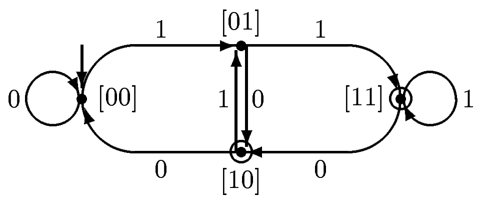
	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow\ \{p\}$	$\{p\}$	$\{p,q\}$
$\{q\}$	$\{r\}$	$\{r\}$
$\{r\}F$	$\varnothing$	$\varnothing$
$\{p,q\}$	$\{p,r\}$	$\{p,q,r\}$
$\{p,r\}F$	$\{p\}$	$\{p,q\}$
$\{q,r\}F$	$\{r\}$	$\{r\}$
$\{p,q,r\}F$	$\{p,r\}$	$\{p,q,r\}$

For example, if we have pebbles on p and q (the fifth row of the table), and if we see input symbol 0 (first column), then in the next step there will be pebbles on p and r. This is because in the automaton N, p is reachable from p under input 0 and r is reachable from q under input 0, and these are the only states reachable from p and q under input 0. The accept states of M (marked F in the table) are those sets containing an accept state of N. The start state of M is $\{p\}$, the set of all start states of N.

Following 0 and 1 transitions from the start state $\{p\}$ of M, one can see that states $\{q,r\}$, $\{q\}$, $\{r\}$, $\varnothing$ of M can never be reached. These states of M are *inaccessible* and we might as well throw them out. This leaves

	0	1
$\rightarrow\ \{p\}$	$\{p\}$	$\{p,q\}$
$\{p,q\}$	$\{p,r\}$	$\{p,q,r\}$
$\{p,r\}F$	$\{p\}$	$\{p,q\}$
$\{p,q,r\}F$	$\{p,r\}$	$\{p,q,r\}$

This four-state automaton is exactly the one you would have come up with if you had built a deterministic automaton directly to remember the last two bits seen and accept if the next to last bit is a 1:



Here the state labels $[bc]$ indicate the last two bits seen (for our purposes the null string is as good as having just seen two 0's). Note that these two automata are isomorphic (*i.e.*, they are the same automaton up to renaming of the states):

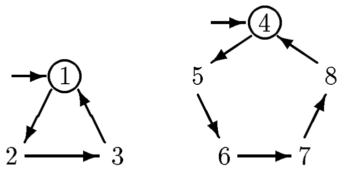
$$\{p\} \approx [00]$$
$$\{p,q\} \approx [01]$$

$$\begin{aligned} \{p,r\} &\approx [10] \\ \{p,q,r\} &\approx [11]\ . \end{aligned}$$

Here is another example. Consider the set

$$\{x \in \{a\}^* \mid |x| \text{ is divisible by 3 or 5}\}\ . \tag{5.1}$$

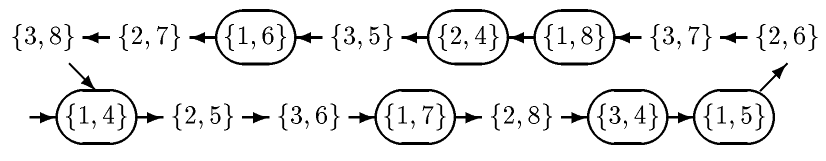
Here is an eight-state nondeterministic automaton N with two start states accepting this set (labels a on transitions are omitted since there is only one input symbol):



The only nondeterminism is in the choice of start state. The machine guesses at the outset whether to check for divisibility by 3 or 5. After that, the computation is deterministic.

Let Q be the states of N. We will build a deterministic machine M whose states are subsets of Q. There are $2^8 = 256$ of these in all, but most will be inaccessible (not reachable from the start state of M under any input). Think about moving pebbles—for this particular automaton, if you start with pebbles on the start states and move pebbles to mark all states the machine could possibly be in, you always have exactly two pebbles on N. This says that only subsets of Q with two elements will be accessible as states of M.

The subset construction gives the following deterministic automaton M with fifteen accessible states:



Next time we will give a formal definition of nondeterministic finite automata and a general account of the subset construction.

Lecture 6

The Subset Construction

Formal Definition of Nondeterministic Finite Automata

A *nondeterministic finite automaton (NFA)* is a five-tuple

$$N = (Q,\ \Sigma,\ \Delta,\ S,\ F)$$

where everything is the same as in a deterministic automaton, except:

- S is a *set* of states, *i.e.* $S \subseteq Q$, instead of a single state. The elements of S are called *start states*;
- Δ is a function

  $$\Delta : Q \times \Sigma \rightarrow 2^Q \ ,$$

  where 2^Q denotes the *power set* of Q, or set of all subsets of Q:

  $$2^Q \stackrel{\text{def}}{=} \{A \mid A \subseteq Q\} \ .$$

Intuitively, $\Delta(p, a)$ gives the set of all states that N is allowed to move to from p in one step under input symbol a. We often write

$$p \xrightarrow{a} q$$

if $q \in \Delta(p, q)$. The set $\Delta(p, a)$ can be the empty set $\varnothing$. The function Δ is called the *transition function*.

Now we define acceptance for NFAs. The function Δ extends in a natural way by induction to a function

$$\widehat{\Delta} : 2^Q \times \Sigma^* \to 2^Q$$

according to the formal definition below. Intuitively, for $A \subseteq Q$ and $x \in \Sigma^*$, $\widehat{\Delta}(A, x)$ is the set of all states reachable under input string x from *some* state in A. Note that Δ takes a single state as its first input and a single symbol as its second input, whereas $\widehat{\Delta}$ takes a *set* of states as its first input and a *string* of symbols as its second input.

$$\widehat{\Delta}(A, \epsilon) \stackrel{\text{def}}{=} A \tag{6.1}$$

$$\widehat{\Delta}(A, xa) \stackrel{\text{def}}{=} \bigcup_{q \in \widehat{\Delta}(A,x)} \Delta(q, a) \ . \tag{6.2}$$

Equation (6.1) says that the set of all states reachable from a state in A under the null input is just A. In (6.2), the notation on the right hand side means the union of all the sets $\Delta(q, a)$ for $q \in \widehat{\Delta}(A, x)$; in other words, $q \in \widehat{\Delta}(A, xa)$ if there exists $r \in \widehat{\Delta}(A, x)$ such that $q \in \Delta(r, a)$.

$$\begin{tikzcd} p \arrow[r, dashed, "x"] & r \arrow[r, "a"] & q \end{tikzcd}$$

Thus $q \in \widehat{\Delta}(A, x)$ if N can move from some state $p \in A$ to state q under input x. This is the nondeterministic analog of the construction of $\widehat{\delta}$ for deterministic automata we have already seen.

Note that for $a \in \Sigma$,

$$\begin{aligned} \widehat{\Delta}(A, a) &= \bigcup_{p \in \widehat{\Delta}(A,\epsilon)} \Delta(p, a) \\ &= \bigcup_{p \in A} \Delta(p, a) \ . \end{aligned}$$

The automaton N is said to *accept* $x \in \Sigma^*$ if

$$\widehat{\Delta}(S, x) \cap F \neq \varnothing \ .$$

In other words, N accepts x if there exists an accept state q (*i.e.*, $q \in F$) such that q is reachable from a start state under input string x (*i.e.*, $q \in \widehat{\Delta}(S, x)$).

We define $L(N)$ to be the set of all strings accepted by N:

$$L(N) = \{x \in \Sigma^* \mid N \text{ accepts } x\} \ .$$

Under this definition, every DFA

$$(Q,\ \Sigma,\ \delta,\ s,\ F)$$

is equivalent to an NFA

$$(Q,\ \Sigma,\ \Delta,\ \{s\},\ F)\ ,$$

where $\Delta(p,a) \stackrel{\text{def}}{=} \{\delta(p,a)\}$. Below we will show that the converse holds as well: every NFA is equivalent to some DFA.

Here are some basic lemmas that we will find useful when dealing with NFAs. The first corresponds to Exercise 3 of Homework 1 for deterministic automata.

Lemma 6.1 *For any $x, y \in \Sigma^*$ and $A \subseteq Q$,*

$$\widehat{\Delta}(A, xy) = \widehat{\Delta}(\widehat{\Delta}(A, x), y)\ .$$

Proof. The proof is by induction on $|y|$. For the basis $y = \epsilon$,

$$\begin{aligned}\widehat{\Delta}(A, x\epsilon) &= \widehat{\Delta}(A, x)\\ &= \widehat{\Delta}(\widehat{\Delta}(A, x), \epsilon) \qquad \text{by (6.1).}\end{aligned}$$

For the induction step,

$$\begin{aligned}\widehat{\Delta}(A, xya) &= \bigcup_{q \in \widehat{\Delta}(A, xy)} \Delta(q, a) && \text{by (6.2)}\\ &= \bigcup_{q \in \widehat{\Delta}(\widehat{\Delta}(A,x),y)} \Delta(q, a) && \text{induction hypothesis}\\ &= \widehat{\Delta}(\widehat{\Delta}(A, x), ya) && \text{by (6.2).}\end{aligned}$$

□

Lemma 6.2 *The function $\widehat{\Delta}$ commutes with set union: for any indexed family A_i of subsets of Q and $x \in \Sigma^*$,*

$$\widehat{\Delta}(\bigcup_i A_i, x) = \bigcup_i \widehat{\Delta}(A_i, x)\ .$$

Proof. By induction on $|x|$. For the basis, by (6.1),

$$\widehat{\Delta}(\bigcup_i A_i, \epsilon) = \bigcup_i A_i = \bigcup_i \widehat{\Delta}(A_i, \epsilon)\ .$$

For the induction step,

$$\begin{aligned}
\widehat{\Delta}(\bigcup_i A_i, xa) &= \bigcup_{p \in \widehat{\Delta}(\bigcup_i A_i, x)} \Delta(p,a) && \text{by (6.2)} \\
&= \bigcup_{p \in \bigcup_i \widehat{\Delta}(A_i, x)} \Delta(p,a) && \text{induction hypothesis} \\
&= \bigcup_i \bigcup_{p \in \widehat{\Delta}(A_i, x)} \Delta(p,a) && \text{basic set theory} \\
&= \bigcup_i \widehat{\Delta}(A_i, xa) && \text{by (6.2).}
\end{aligned}$$

□

In particular, expressing a set as the union of its singleton subsets,

$$\widehat{\Delta}(A,x) = \bigcup_{p \in A} \widehat{\Delta}(\{p\}, x) \ . \tag{6.3}$$

The Subset Construction: General Account

The subset construction works in general. Let

$$N = (Q_N,\ \Sigma,\ \Delta_N,\ S_N,\ F_N)$$

be an arbitrary NFA. We will use the subset construction to produce an equivalent DFA. Let M be the DFA

$$M = (Q_M,\ \Sigma,\ \delta_M,\ s_M,\ F_M) \ ,$$

where

$$\begin{aligned}
Q_M &\overset{\text{def}}{=} 2^{Q_N} \\
\delta_M(A,a) &\overset{\text{def}}{=} \widehat{\Delta}_N(A,a) \\
s_M &\overset{\text{def}}{=} S_N \\
F_M &\overset{\text{def}}{=} \{A \subseteq Q_N \mid A \cap F_N \neq \varnothing\} \ .
\end{aligned}$$

Note that δ_M is a function from states of M and input symbols to states of M, as it should be, because states of M are *sets* of states of N.

Lemma 6.3 *For any $A \subseteq Q_N$ and $x \in \Sigma^*$,*

$$\widehat{\delta}_M(A,x) = \widehat{\Delta}_N(A,x) \ .$$

Proof. Induction on $|x|$. Basis $x = \epsilon$: we want to show

$$\widehat{\delta}_M(A, \epsilon) = \widehat{\Delta}_N(A, \epsilon) \ .$$

But both of these are A, by definition of $\widehat{\delta}_M$ and $\widehat{\Delta}_N$.

Induction step: assume that

$$\widehat{\delta}_M(A, x) = \widehat{\Delta}_N(A, x) \ .$$

We want to show the same is true for xa, $a \in \Sigma$.

$$\begin{aligned}
\widehat{\delta}_M(A, xa) &= \delta_M(\widehat{\delta}_M(A, x), a) && \text{definition of } \widehat{\delta}_M \\
&= \delta_M(\widehat{\Delta}_N(A, x), a) && \text{induction hypothesis} \\
&= \widehat{\Delta}_N(\widehat{\Delta}_N(A, x), a) && \text{definition of } \delta_M \\
&= \widehat{\Delta}_N(A, xa) && \text{Lemma 6.1.}
\end{aligned}$$

□

Theorem 6.4 *The automata M and N accept the same set.*

Proof.

$$\begin{aligned}
& x \in L(M) \\
\iff & \widehat{\delta}_M(s_M, x) \in F_M && \text{definition of acceptance for } M \\
\iff & \widehat{\Delta}_N(S_N, x) \cap F_N \neq \varnothing && \text{definition of } s_M \text{ and } F_M\text{, Lemma 6.3} \\
\iff & x \in L(N) && \text{definition of acceptance for } N.
\end{aligned}$$

□

ϵ-Transitions

Here is another extension of finite automata that turns out to be quite useful, but really adds no more power.

An *ϵ-transition* is a transition with label ϵ, a letter that stands for the null string ϵ:

$$p \xrightarrow{\epsilon} q$$

The automaton can take such a transition anytime without reading an input symbol.

Example 6.5

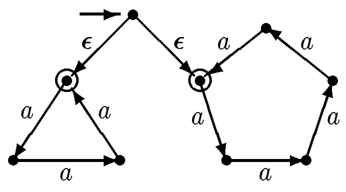
$$\begin{tikzcd}
{} \arrow[r] & s \arrow[r, "\epsilon"] \arrow[d, "b"] & t \arrow[r, "\epsilon"] \arrow[d, "b"] & u \arrow[d, "b"] \\
& p \arrow[ur, "\epsilon"'] & q \arrow[ur, "\epsilon"'] & \textcircled{r}
\end{tikzcd}$$

If the machine is in state s and the next input symbol is b, it can nondeterministically decide to do one of three things:

- read the b and move to state p;
- slide to t without reading an input symbol, then read the b and move to state q; or
- slide to t without reading an input symbol, then slide to u without reading an input symbol, then read the b and move to state r.

The set of strings accepted by this automaton is $\{b, bb, bbb\}$. □

Example 6.6 Here is a nondeterministic automaton with ϵ-transitions accepting the set $\{x \in \{a\}^* \mid |x| \text{ is divisible by 3 or 5}\}$:



The automaton chooses at the outset which of the two conditions to check for (divisibility by 3 or 5) and slides to one of the two loops accordingly without reading an input symbol. □

The main benefit of ϵ-transitions is convenience. They do not really add any power: a modified subset construction involving the notion of *ϵ-closure* can be used to show that every NFA with ϵ-transitions can be simulated by a DFA without ϵ-transitions (Miscellaneous Exercise 9), thus all sets accepted by nondeterministic automata with ϵ-transitions are regular. We will also give an alternative treatment in Lecture 10 using homomorphisms.

More Closure Properties

Recall that the concatenation of sets A and B is the set

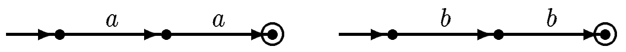
$$AB = \{xy \mid x \in A \text{ and } y \in B\} .$$

For example,

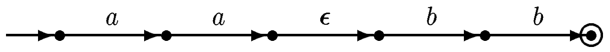
$$\{a, ab\}\{b, ba\} = \{ab, aba, abb, abba\} \ .$$

If A and B are regular, then so is AB. To see this, let M be an automaton for A and N an automaton for B. Make a new automaton P whose states are the union of the state sets of M and N, and take all the transitions of M and N as transitions of P. Make the start states of M the start states of P and the final states of N the final states of P. Finally, put ϵ-transitions from all the final states of M to all the start states of N. Then $L(P) = AB$.

Example 6.7 Let $A = \{aa\}$, $B = \{bb\}$. Here are automata for A and B:



Here is the automaton you get by the construction above for AB:



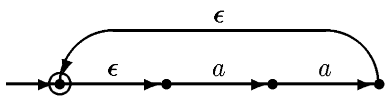
□

If A is regular, then so is its asterate

$$\begin{aligned} A^* &= \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots \\ &= \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\} \ . \end{aligned}$$

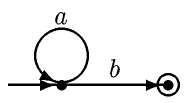
To see this, take an automaton M for A. Build an automaton P for A^* as follows. Start with all the states and transitions of M. Add a new state s. Add ϵ-transitions from s to all the start states of M and from all the final states of M to s. Make s the only start state of P and also the only final state of P (thus the start and final states of M are *not* start and final states of P). Then P accepts exactly the set A^*.

Example 6.8 Let $A = \{aa\}$. Consider the three-state automaton for A in Example 6.7. Here is the automaton you get for A^* by the construction above:



□

In this construction, you must add the new start/final state s. You might think that putting in ϵ-transitions from the old final states back to the old start states and making the old start state a final state should suffice, but this doesn't always work. Here's a counterexample:



The set accepted is $\{a^n b \mid n \geq 0\}$. The asterate of this set is

$$\{\epsilon\} \cup \{\text{strings ending with } b\} ,$$

but if you put in an ϵ-transition from the final state back to the start state and made the start state a final state, then the set accepted would be $\{a, b\}^*$.

Historical Notes

Rabin and Scott [102] introduced nondeterministic finite automata and showed using the subset construction that they were no more powerful than deterministic finite automata.

Closure properties of regular sets were studied by Ginsburg and Rose [48, 46], Ginsburg [43], McNaughton and Yamada [85], and Rabin and Scott [102], among others.