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 some fundamental and pervasive concepts along the way: *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 with 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 [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 attempted 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;

(ii) context-free grammars;

(iii) unrestricted grammars.

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), (ii), and (iii) are equivalent in computational power to the machine models (i), (ii), and (iii) above, respectively. There is even a fourth natural class called the *context-sensitive* grammars and languages, which fits in between (ii) and (iii) 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 focuses 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 twentieth 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 twentieth century mathematics. 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 in terms of their mathematical 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 Versus 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, and
- 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 have to deal with only 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 $a, b, c, \ldots$. We usually 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:

$$a^0 \stackrel{\text{def}}{=} \epsilon,$$
$$a^{n+1} \stackrel{\text{def}}{=} a^n a.$$

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

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

□

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 see 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. Here are some useful properties of concatenation.

- 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 see some other examples later 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:

$$x^0 \stackrel{\text{def}}{=} \epsilon,$$
$$x^{n+1} \stackrel{\text{def}}{=} x^n x.$$

- 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 $\#1(00000) = 0$.

- A *prefix* of a string x is an initial substring of x; that is, a string y for which there exists a string z such that $x = yz$. For example, *abaab* is a prefix of *abaababa*. The null string is a prefix of every string, and

every string is a prefix of itself. A prefix y of x is a *proper* prefix of x if $y \neq \epsilon$ and $y \neq x$. □

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:

$$A^0 \stackrel{\text{def}}{=} \{\epsilon\},$$
$$A^{n+1} \stackrel{\text{def}}{=} AA^n.$$

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,

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

Also,

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

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

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

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 that 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*:

$$(A \cup B) \cup C = A \cup (B \cup C),$$
$$(A \cap B) \cap C = A \cap (B \cap C),$$
$$(AB)C = A(BC).$$

- Set union and set 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

$$A(\bigcup_{i \in I} B_i) = \bigcup_{i \in I} AB_i,$$
$$(\bigcup_{i \in I} B_i)A = \bigcup_{i \in I} B_i A.$$

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:

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

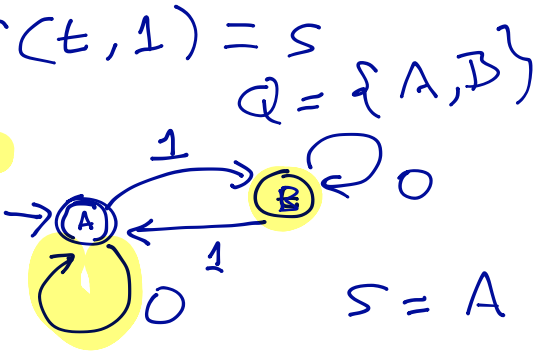
- The asterate operation * satisfies the following properties:

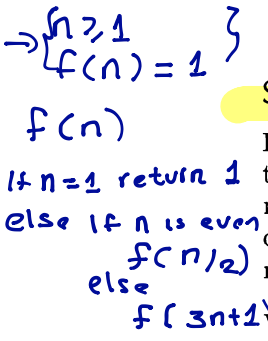
HW

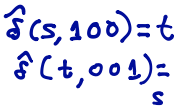
$$A^*A^* = A^*,$$
$$A^{**} = A^*,$$
$$A^* = \{\epsilon\} \cup AA^* = \{\epsilon\} \cup A^*A,$$
$$\varnothing^* = \{\epsilon\}.$$

Lecture 3

Finite Automata and Regular Sets







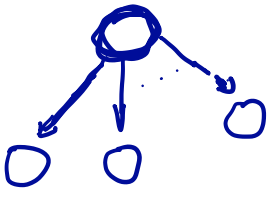
States and Transitions

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

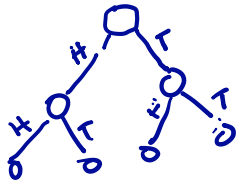
We assume that state transitions are instantaneous. This is a mathematic abstraction. In reality, transitions usually take time. Clock cycles in dig tal computers enforce this abstraction and allow us to treat computers a digital instead of analog devices.

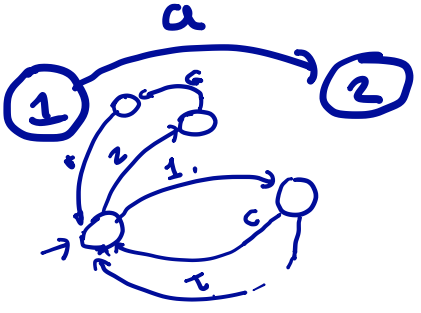
There are innumerable examples of state transition systems in the re world: electronic circuits, digital watches, elevators, Rubik's cube (54!/9 states and 12 transitions, not counting peeling the little sticky squares off the game of Life (2^k states on a screen with k cells, one transition).

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



rm *.pdf





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 that $Q \times \Sigma$ 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 following example.

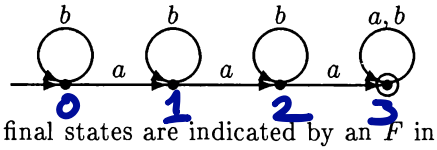
Example 3.1 Here is an example of a simple four-state finite automaton. We'll take the set of states to be $\{0, 1, 2, 3\}$; the input alphabet to be $\{a, b\}$; the start state to be 0; the set of accept states to be $\{3\}$; and the transition function to be

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

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

		a	b
→	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 →. 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 $p \in F$ and *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, beginning in its start state 0, will be in state 3 after scanning the input string *baabbaab*, so that string is accepted; however, 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, state 1 if it has seen one a, state 2 if it has seen two a's, and state 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

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

from δ by induction on the length of x:

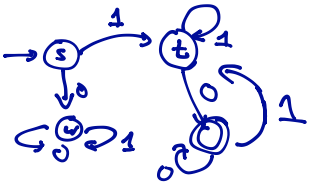
$$\widehat{\delta}(q, \epsilon) \stackrel{\text{def}}{=} q, \tag{3.1}$$

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

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 multistep 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:

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



$$= \delta(q, a) \qquad \text{by (3.1).}$$

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

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

and *rejected* by the automaton M if

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

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 $L(M)$:

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

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

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

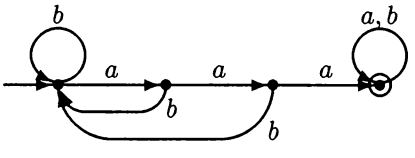
so this is a regular set.

Example 3.2 Here is another example of a regular set and a finite automaton accepting it. Consider the set

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

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 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
⋮	⋮

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

		0	1
→	**0** *F*	**0**	**1**
	1	**2**	**0**
	2	**1**	**2**