Our textbook is the sixth edition of *Formal Languages and Automata*, by Peter Linz.

Alphabets, Strings, Languages, and Machines

Alphabets

An *alphabet* is a finite set of *symbols*. There is no definition of *symbol.* Alphabets used in this course include:
The alphabet of all ASCII symbols.
The Roman alphabet: upper case, lower case, or both.
The decimal alphabet: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
The binary alphabet: $\{0, 1\}$.
The unary alphabet: $\{1\}$.
Small subsets of the Roman alphabet, such as $\{a, b\}$.

Strings

A *string* is a finite sequence of symbols over some alphabet. For example, if $\Sigma = \{a, b, c\}$, then a, b, $aba, abccaa$, are strings of length 1, 3, or 6 over $\{a, b, c\}$. The empty string, denoted λ (or ϵ) has length zero and consists of no symbols.

We write Σ^* to mean the set of all strings over the alphabet Σ. Σ^*, which is countably infinite. For any string $w \in \Sigma^*$, we let $|w|$ be the length of w.

The binary alphabet is of particular importance in computer science. We use the term *binary string* to mean any string over the binary alphabet.

Languages

A *language* is defined to be a set of strings over a particular alphabet. If L is a language over Σ, then $L \subseteq \Sigma^*$.

There is no definition of symbol, and thus anything can be a symbol. The language of DNA strings is over the alphabet consisting of the four nucleotides: adenine, thymine, guanine, and cytosine, usually abbreviated as A, T, G, and C.

Example. A *programming language* is a set of *programs*, each of which is a string over the alphabet consisting of all symbols used in that language. including blank and end-of-line.

A common claim is that languages are used for communication. This is true in many cases, but it is not part of the definition.

We do not deal with natural languages, such as English, in this course.

Numerals and Numbers

We distinguish between a number and a numeral. A number is an abstract object which has no physical existence. A numeral is something (usually a string) which denotes a number. If n is a number, we write $\langle n\rangle$ to mean a numeral which denotes n.

Problems and Languages

We are primarily interested in infinite problems, that is, problems which have infinitely many instances. For example, "What is 2+3?" is an instance of the addition problem.

A 0/1 problem is any problem where the answer for each instance is either 0 (false) or 1 (true). For example, an instance of the *primality* problem is a numeral $\langle n\rangle$, and the answer is 1 (true) if n is prime, 0 (false) otherwise.

A problems that is not 0/1 could have a 0/1 version. For example, instead of asking for the prime factors of n, we could ask whether n has a prime factor smaller than a given other number a.

Languages and 0/1 problems are essentially the same thing. For any language L, there is a *membership problem.* If $L \subseteq \Sigma^*$, every string over Σ is an instance of the membership problem for L. For the instance $w \in \Sigma^*$, the answer is 1 if $w \in L$ and 0 if $w \notin L$. Many language classes, such as $\mathcal{P}$-TIME, are defined by the hardness of their membership problems. A language is said to be "hard" or "easy" if its membership problem is hard or easy. The precise meaning depends on the context of the discussion. Here are some examples that arise in this course.

Easy	Hard
Regular	Non-Regular
Polynomial Time	$\mathcal{NP}$-Hard
Recursive	Undecidable

Machines

A *machine* in this course is an *abstract machine,* which is a mathematical object. (The computer on your desk is a *physical* machine.) A *computation* of a machine is a sequence of steps. A machine has an initial *configuration,* also called the *instanteous description,* or **id**. There is an initial **id**, and at each step, the instantaneous description changes according to the rules of the machine. A computation can be infinite, or end with a halt, or the machine may *hang*, meaning there is no legal next step. Each **id** can be desribed by a string. This string must encode everything needed for the computation, such as the machine's current state, contents of its memory, unread input, and written output. A string is necessarily finite, but during an infinite computation, the **id** could increase its length without limit.

Accept and Decide

We say that a non-deterministic machine M *accepts* a string w if, given the input w, M may halt in an *accepting* state. We say language L if M accepts every $w \in L$ and does not accept any string not in L.

We say that M *decides* L if, given an input string w, M halts in an accepting state if $w \in L$ and halts in a rejecting state if $w \notin L$.[1]

Regular Languages

The *Chomsky hierarchy* is consists of four types (or classes) of languages. The easiest of these is the class of Type 3 languages, otherwise known as *regular* languages.

Deterministic Finite Automata

A machine M is called a *finite automaton* (FA) if its **id** consists of one of a finite set of states together with its current unread input. A *deterministic finite automaton* (DFA) M has a finite set of *states* Q, one of which (usually called q_0) is the *start* state. There is a subset $F \subseteq Q$ of *final* states. An input for a DFA is a string $w \in \Sigma^*$, where Σ is called the input alphabet. M also has a *transition function* $\delta : Q \times \Sigma \to Q$. Formally, M is the quintuple $(Q, \Sigma, \delta, q_0, F)$. An **id** of M is an ordered pair (q, u), where $q \in Q$ is the current state and $u \in \Sigma^*$ is the remaining (unread) input. The initial **id** of M is (q_0, w), where w is the input string. We can generalize the transition function to $\delta : Q \times \Sigma^* \to Q$ by recursion:

$\delta(q, \lambda) = q$, for $q \in Q$.

$\delta(q, wa) = \delta(\delta(q, w), a)$, for $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

Steps of M. The number of steps a DFA M takes during a computation is equal to the length of the input string. During each step, M reads the first symbol of the remaining input, then changes its state. If $q \in Q$ is the current state and a is the next symbol of input, the state changes to $\delta(q, a)$). If the last state is final, w is accepted, otherwise w is rejected. If a DFA M accepts a language L, it is also true that M decides L, since it always halts. A language is defined to be *regular* if it is accepted by some DFA

Example

Let M be the DFA where $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, and δ is defined by the transition table given in Table 1, and illustrated as a state diagram in Figure 2

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_1

Table 1

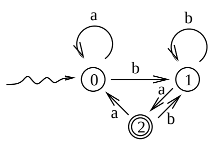

Figure 2: State Diagram of M

Figure 3 shows a computation of M which accepts the string *abba*, while Figure 4 shows a computation of M which rejects the string *abab*.

[1]If M accepts L, there there is no requirement that it actually reject a string $w \notin L$. It could instead hang or run forever.

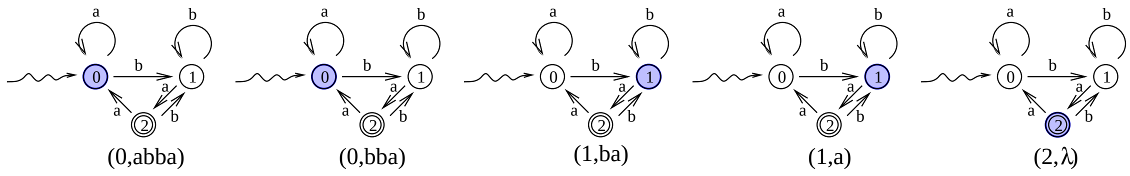

Figure 3: Computation of M accepting *abba*. For simplicity, the states are labeled 0, 1, 2 instead of q_0, q_1, q_2. The final state is doubly circled. The figures show the sequence of **ids**. The current state is indicated in blue, and the current **id** is underneath the figure. Note that the last state is final.

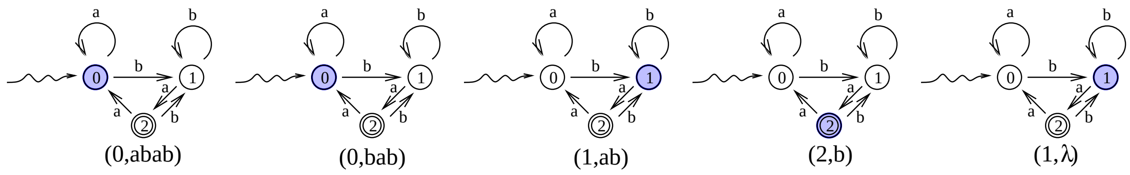

Figure 4: Computation of M rejecting *abab*. Note that the last state is not final.

Dead States. A DFA may have a *dead state*, a state which is not final, and which the machine cannot leave, regardless of the remaining inputs. The machine shown in Figures 3 and 4 does not have a dead state. The machine shown in Figure 5 has a dead state, q_1.

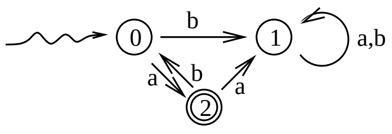

Figure 5: A DFA with one dead state.

Non-deterministic Finite Automata

A machine is *deterministic* if, from a given **id**, there is at most one possible next **id** A machine is *non-determinisiic* if, from any given **id**, there can be any number of possible next **ids**. Confusingly, any deterministic machine is a non-deterministic machine.

Recall that, for any set S, 2^S, the *powerset* of S, is the set of all subsets of S. If $|S| = n$, then $\left|2^S\right| = 2^n$.

A non-deterministic finite state automaton (NFA) has the same basic parts as a DFA, except that, for clarity, we write Δ for the transition function, as is done in Wikipedia. For any state $q \in Q$ and $a \in \Sigma$, $\Delta(q, a)$ is a set of states rather than a single state.[2] Using recursion, we also define $\Delta(q, w) \subseteq Q$ for $q \in Q$, $w \in \Sigma^*$:
$\Delta(q, \lambda) = \{q\}$.
$\Delta(q, wa) = \bigcup_{q' \in \Delta(q,w)} \Delta(q', a)$.

[2]Our textbook still uses δ for an NFA.

If the current state is q and the next input is a, then the machine can move to any member of $\Delta(q, a)$. An NFA may also have the option of changing states without reading a symbol. Such a move is called a *λ-move* or an *ϵ-move*. Formally, an NFA is a quintuple $(Q, \Sigma, \Delta, q_0, F)$ where $\Delta : \Sigma \cup \{\lambda\} \times Q \to 2^Q$, $q_0 \in Q$, and $F \subseteq Q$.

Example. Let M be the NFA whose state diagram is shown in Figure 6.

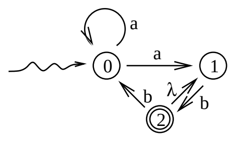

Figure 6: NFA M

Δ	a	b	λ
q_0	$\{q_0, q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_0\}$	$\{q_1\}$

Table 7: Transition Table of M

NFA Steps. The initial **id** of an NFA is the ordered pair (q_0, w), where w is the input string. During each step, either M reads a string and changes state, or uses a λ-move to change states and read nothing.

We show a computation of M with input abb in Figure 8. At the first step, M reads a and moves to q_1. Alternatively, M could read a and stay in q_0, but then it would be impossible to accept the input. In case of a choice, an NFA always make a choice which leads to acceptance, if that is possible.

At the third step, M has another choice. M makes the correct guess, namely to make a λ-move, reading nothing and changing to state q_1. This choice allows the input to be accepted at the next step.

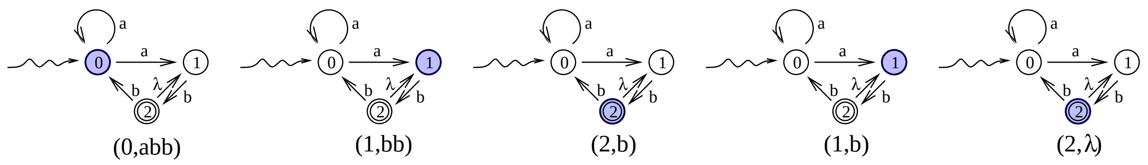

Figure 8: Accepting Computation of M with Input abb.

For a given input, the number of possible computations of an NFA could be an exponential function of the length of the input string. Of these computations, there could be some that end in a final state, some that end in a non-final state, and some that never finish reading the input, either by hanging or entering an infinite loop. When we say that M accepts w, we mean that there is at least one computation of M starting at (q_0, w) which ends in a final state. An NFA always make the correct guess, if there is one, to achieve acceptance. This rule, "benevolent non-determinishm," also holds for other non-deterministic machines that we study, such as push-down automata (PDA) and non-deterministic Turing machines (NTM).

Equivalent Machines

Informally, machines M_1 and M_2 are *equivalent* if the do the same thing. For example, two finite automata are equivalent of they accept the same language. The number of steps of a computation does not play a role in this definition; the number of steps required by two equivalent machines with the same input could be different. If M is a finite automaton, there could be many other automata equivalent to M; however, the minimal DFA for a regular language is unique, as stated in Theorem 1.

Theorem 1 *If L is a regular language, there is a unique minimal DFA which accetps L.*

Minimal means smallest number of states. If M_1 is a minimal DFA which accepts L and M_2 is also a minimal DFA which accepts L, the state diagrams for the two machines are identical, expect for possibly changing the names of the states.

Minimizing a DFA

We now give Hopcroft's algorithm for finding a minimal DFA. Let M be a DFA which accepts a language L over an alphabet Σ. The algorithm consists of two parts:

(a) Elimination of useless states.

(b) Identification of equivalent (indistinguishable) states.

Useless States. Supose $M = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton. A state q_k of M is defined to be *useless* if no computation of M ever reaches state q_k.

Informally, two states q_i, q_j are *equivalent* if, after reading some prefix of the input string, it doesn't matter whether a computation is in q_i or q_j. More formally, we say that q_i and q_j are *distinguished* if one of the following holds:

(a) $q_i \in F$ and $q_j \notin F$,

(b) $q_i \notin F$ and $q_j \in F$,

(c) For some $a \in \Sigma$, $\delta(q_i, a)$ and $\delta(q_j, a)$ are distinguished. Note that the definition of distinguished is recursive. Finally, $q_i, q_j \in Q$ are indistinguishable, that is, equivalent, if they are not distinguished.

Example 1. Let M be the DFA illustrated by the state diagram in Figure 9.

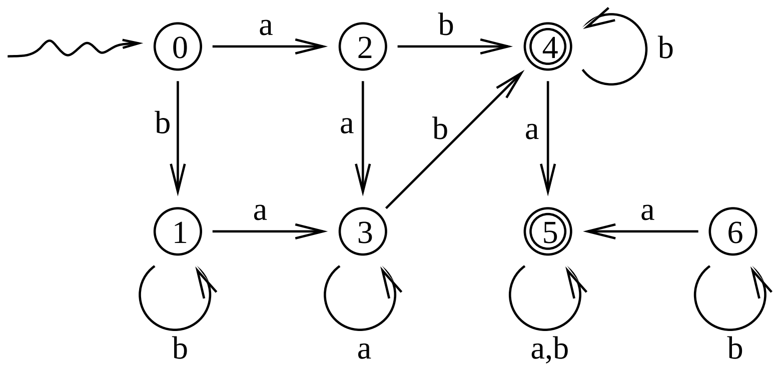

Figure 9: The DFA M

We first note that q_6 is useless, hence we delete it. We write a square array whose rows and columns are the remaining states. We mark the entry in row q_i and column q whenever we prove that those two states are distinguished. Initially, no final state is equivalent to any non-final state.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0					$\times$	$\times$
q_1					$\times$	$\times$
q_2					$\times$	$\times$
q_3					$\times$	$\times$
q_4	$\times$	$\times$	$\times$	$\times$		
q_5	$\times$	$\times$	$\times$	$\times$		

We now iterate through $Q \times \Sigma \times Q$. For each (q_i, a, q_j), we mark the $(i, j)^{\text{th}}$ entry of the array if, for some $x \in \Sigma$, we can determine that $\delta(q_i, x)$ and $\delta(q_j, x)$ are distinguished.
We first note that $\delta(q_0, b) = q_1$ and $\delta(q_2, b) = q_4$, which are distiguished. Thus q_0 and q_2 are distinguished.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0			$\times$		$\times$	$\times$
q_1					$\times$	$\times$
q_2	$\times$				$\times$	$\times$
q_3					$\times$	$\times$
q_4	$\times$	$\times$	$\times$	$\times$		
q_5	$\times$	$\times$	$\times$	$\times$		

Similarly, we can determine that the pairs (q_0, q_3), (q_1, q_2), and (q_1, q_3) are distinguished, and we mark the array accordingly.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0			$\times$	$\times$	$\times$	$\times$
q_1			$\times$	$\times$	$\times$	$\times$
q_2	$\times$	$\times$			$\times$	$\times$
q_3	$\times$	$\times$			$\times$	$\times$
q_4	$\times$	$\times$	$\times$	$\times$		
q_5	$\times$	$\times$	$\times$	$\times$		

We iterate over $Q \times \Sigma \times Q$ until no further pairs are found to be distinguished. All unmarked pairs are then equivalent. We identify the equivalent pairs (q_0, q_1), (q_2, q_3), and (q_4, q_5).

The resulting minimal DFA equivalent to M is illustrated in Figure 10.

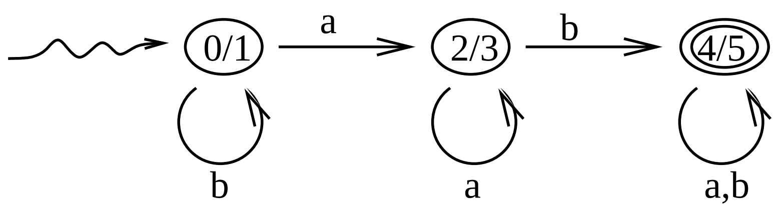

Figure 10: The minimal DFA equivalent to M.

Example 2. Let M be the DFA illustrated by the state diagram in Figure 11.

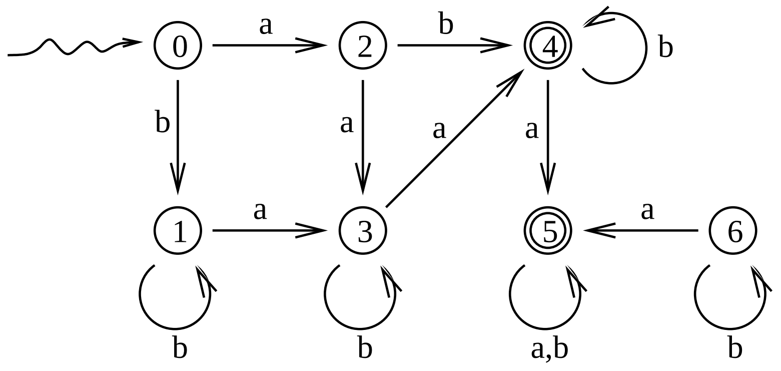

Figure 11: The DFA M

We first note that q_6 is useless, hence we delete it. We write a square array whose rows and columns are the remaining states. We mark the entry in row q_i and column q whenever we prove that those two states are distinguished. Initially, no final state is equivalent to any non-final state.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0					×	×
q_1					×	×
q_2					×	×
q_3					×	×
q_4	×	×	×	×		
q_5	×	×	×	×		

We now iterate through $Q \times \Sigma \times Q$. For each $x \in \Sigma$ and $q_i, q_j \in Q$, we mark the $(i,j)^{\text{th}}$ entry of the array if we can determine that $\delta(q_i, x)$ and $\delta(q_j, x)$ are distinguished.

During the first iteration, q_0 and q_1 are not distinguished, since q_2 and q_3 are not distinguished.

Continuing the first iteration, we can distinguish q_2 q_0 from q_2, since $\delta(q_0, b) = q_1$, which is distinguished from $\delta(q_2, b) = q_4$. Similarly, we distinguish q_0 from q_3, q_1 from q_2, and q_1 from q_3.
Continuing the first iteration, we distinguish q_2 and q_3, since $\delta(q_2, a) = q_3$, which is distingused from $\delta(q_3, a) = q_4$.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0					×	×
q_1					×	×
q_2				×	×	×
q_3			×		×	×
q_4	×	×	×	×		
q_5	×	×	×	×		

Similarly, we can determine that the pairs (q_0, q_3), (q_1, q_2), and (q_1, q_3) are distinguished, and we mark the array accordingly.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0			×	×	×	×
q_1			×	×	×	×
q_2	×	×		×	×	×
q_3	×	×	×		×	×
q_4	×	×	×	×		
q_5	×	×	×	×		

During the second iteration, we can distinguish q_0 and q_1, since $\delta(q_0, a) = q_2$ which is now distinguished from $\delta(q_1, a) = q_3$.

	q_0	q_1	q_2	q_3	q_4	q_5
q_0		×	×	×	×	×
q_1	×		×	×	×	×
q_2	×	×		×	×	×
q_3	×	×	×		×	×
q_4	×	×	×	×		
q_5	×	×	×	×		

Continuing to iterate over $Q \times \Sigma \times Q$, no further pairs are found to be distinguised. All unmarked pairs are then equivalent. We identify the equivalent pair (q_4, q_5). The resulting minimal DFA equivalent to M is illustrated in Figure 12.

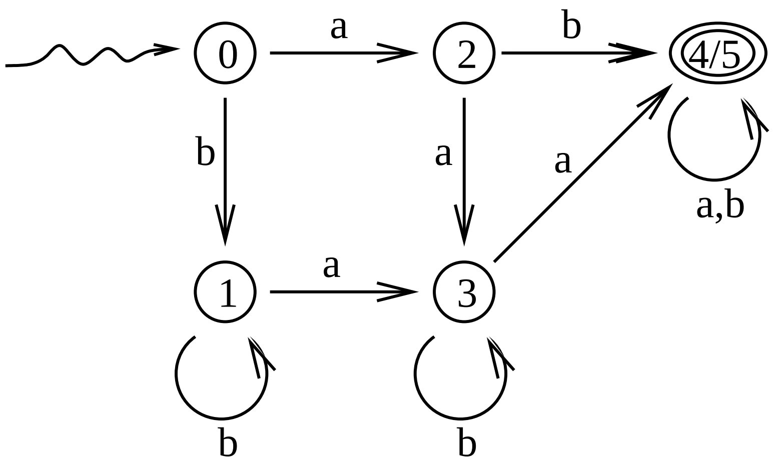

Figure 12: The minimal DFA equivalent to M.

NFA to DFA

Given an NFA M_1 with n states, the Rabin-Scott powerset construction yields an equivalent DFA M_2 with 2^n states. We can then apply Hopcroft's algorithm to obtain a minimal DFA, which may have fewer states.

Let $M_1 = (Q, \Sigma, \Delta, q_0, F)$ be an NFA. We first consider the case where M_1 no λ-transitions.

Let $\mathcal{F} = (\mathcal{Q} \subseteq Q : \mathcal{Q} \cap F \neq \emptyset)$. Let $M_2 = \left(2^Q, \Sigma, \delta, \{q_0\}, \mathcal{F}\right)$, where $\delta(a, \mathcal{Q}) = \bigcup_{q \in \mathcal{Q}} \Delta(q, a)$ for all $\mathcal{Q} \subseteq Q$, and $\mathcal{F} = (\mathcal{Q} \subseteq Q : \mathcal{Q} \cap F \neq \emptyset)$. Then M_2 is equivalent to M_1.

If there are any λ-transitions, we first replace the initial NFA by its λ-closure. The transitive closure of an NFA is obtained by removing λ-transitions one at a time, modifying F or δ at each step, according the following rules.

1. Pick a λ-transition from q_i to q_j.
    2. If q_j is final and q_i is not, make q_i final.
        3. If $q_k \in \delta(a, q_j)$ for some $a \in \Sigma \cup \{\lambda\}$, let q_k become a member of $\delta(q_i, a)$.
        4. Repeat step 3 until there are no further changes.
    5. Delete the λ-transition from q_i to q_j.

6. Return to step 1 if there are any more λ-transitions.

Figure 13 shows the complete calculation of a minimal DFA equivalent to an NFA.

Figure 13(a) shows the NFA M_1.

Figure 13(b) shows the NFA after the λ-transitions from q_0 to q_2 and from q_3 to q_4 are removed.

Figure 13(c) shows the NFA after the λ-transition from q_1 to q_3 is removed, yielding the λ-closure of M_2.

Figure 13(d) is obtained by deleting the now useless state q_4.

Figure 13(e) shows the NFA obtained by the powerset construction. There should be 2^4 states. Eleven of those are not shown since they are useless. (The usual braces for the subsets are not shown.)

States $\{q_0\}$ and $\{q_0, q_2\}$ are indistinguished, and are hence identified in the minimal DFA M_2, shown in Figure 13(f).

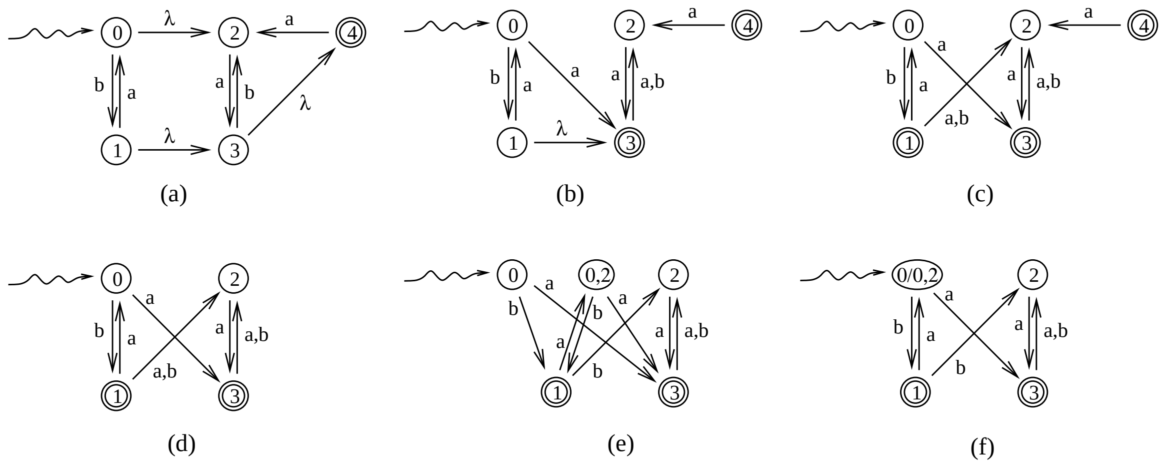

Figure 13: Construction of a Minimal DFA Equivalent to an NFA.

Regular Expressions

A regular expression is an algebraic expression that defines, or describes, a regular language. Regular expressions make use of *closure properties* of the class of regular languages.

Closure Properties of the Class of Regular Languages

We define operations on langagues.

1. (Union) If L_1 and L_2 are regular languages, their union $L_1 \cup L_2$, usually written $L_1 + L_2$, is a regular language.

2. (Intersection) If L_1 and L_2 are regular languages, their intersection $L_1 \cap L_2$ is a regular language.

3. (Complement) If L is a language over an alphabet Σ, the *complement* of L is the set of all strings

over Σ which are not members of L. We write $\Sigma^*\backslash L$ or L' for the complement of L. The complement of any regular language is regular.

4. (Concatenation) If L_1 and L_2 are languages, their concatenation is $L_1L_2 = \{uv : u \in L_1, v \in L_2\}$. If L_1 and L_2 are regular, L_1L_2 is regular. We define "powers" of a language by repeated concatentation. $L^2 = LL$, $L^3 = LLL$, and so forth. $L^1 = L$ and $L^0 = \{\lambda\}$, the language consisting of just one string, the empty string.

5. (Kleene Closure) We define the *Kleene closure* of a language L to be the union of all powers of L, written L^*. We can write $L^* = L^0 + L^1 + L^2 + \cdots$ Formally, a string is in L^* if it is the concatenation of finitely many members of L. It is important to note that the empty string is a always a member of L^*. If L is regular, L^* is regular.

Exercise 1 Let $L_1 = \{a, ab, c\}$ and $L_2 = \{\lambda, a, b\}$.
(a) Find $L_1 + L_2$. Ans: $\{\lambda, a, b, c, ab\}$
(b) Find $L_1 \cap L_2$. Ans: $\{a\}$
(c) Find L_1L_2. Ans: $\{a, c, aa, ca, cb, aba, abb\}$
(d) Draw a state diagram for a DFA which accepts the complement of the language accepted by the DFA in Figure 2.

Ans: Simply invert the DFA defined in Table 1 whose state diagram is given by Figure 2. meaning that every final state becomes non-final and every non-final state becomes final. (This trick does not work for an NFA.)

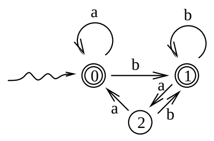

Figure 14: State Diagram of a DFA which Accepts the Complement of the Language Accepted by the DFA whose State Diagram is Shown in Figure 2.

Regular Expressions

Every regular expression describes a regular language; conversely, every regular language L is described by a regular expression, but different regular expressions can describe the same language. If Σ is the alphabet of L, a regular expression for L is a string over the alphabet $\Sigma + \{\lambda, \not\phi, +, ^*, (,)\}$.

Regular expressions for languages over Σ are algebraic expressions, where the variables are the symbols of Σ, together with λ and $\not\phi$; and the operators are union, represented by "+", concatenation, represented by concatenation, and Kleene closure, represented by *. Among these operators, Kleene closure has highest precedence, followed by concatenation, followed by union. Parentheses override precedence in the usual manner.

If $a \in \Sigma$, the regular expression a represents the language $\{a\}$. The regular expression λ represents the language $\{\lambda\}$, and the regular expression $\not\phi$ represents the empty language, $\emptyset$.

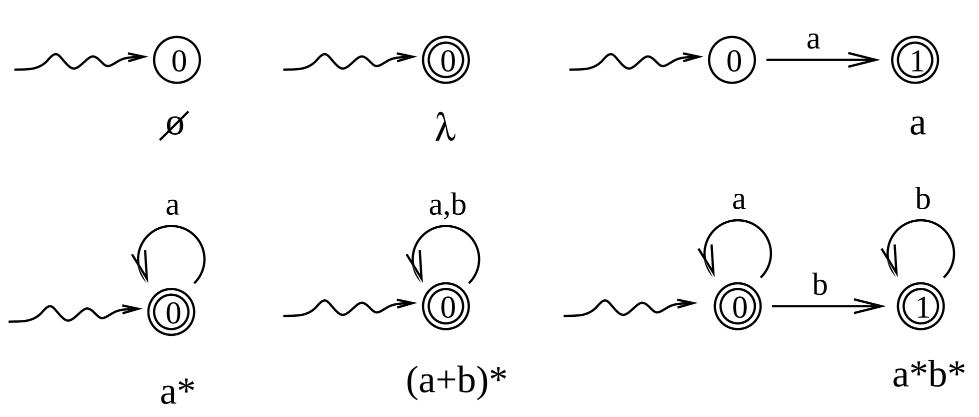

Figure 15: Regular Expressions over $\Sigma = \{a, b\}$ with Equivalent DFA.

Combining Regular Expressions

We now show how to find a regular expression for the union, concatenation, or Kleene closure of languages which already have regular epressions. In the list below, we use parentheses to ensure that operations are done in the right order.

(a) If u is a regular expression for a regular language L, then (u) is also regular expression for L. (b) If u, v are regular expressions for regular languages L and M, then $u + v$ is a regular expression for the union $L + M$. (c) If u, v are regular expressions for regular languages L and M, then $(u)(v)$ is a regular expression for the concatenation LM. One of both of those pairs of parentheses my be unnecessary. (d) If u is regular expressions for a regular language L, then $(u)^*$ is a regular expression for the Kleene closure L^*. The pair of parentheses may be unnecessary.

In Figure 15 we show a DFA equivalent to six simple regular expressions.

Union is commutative, associate and idempotent, but concatenation is only associative. For example, $a + b = b + a$, $(a + b) + c = a + (b + c)$, $a + a = a$, and $a(bc) = a(bc)$,

Concatenation distibutes over union on both sides; for example, $a(b + c)d = abd + acd$. Kleene closure does not distribute over concatenation. For example, $(ab)^* \neq a^*b^*$. Kleene closure is also idempotent: for example, $(a^*)^* = a^*$.

T/F Questions about Regular Expressions

(a) ---- $\lambda^* = \lambda$
(b) ---- $a + \lambda = a$
(c) ---- $a + \emptyset = a$
(d) ---- $\emptyset^* = \emptyset$
(e) ---- $\emptyset^* = \lambda$
(f) ---- $\emptyset(a + b) = \emptyset$
(g) ---- $\emptyset(a + b) = a + b$
(h) ---- $ab^* + ab^* = ab^*$
(i) ---- $(a + ab) = a(b + \lambda)$

Answers to Questions:

(a) T
(b) F
(c) T
(d) F
(e) T
(f) T
(g) F
(h) T
(i) T

Five Definitions of a Regular Language. The follwing five definitions of a regular language are equivalent:
1. A language is regular if and only if it is accepted by some DFA.
2. A language is regular if and only if it is accepted by some NFA.
3. A language is regular if and only if it is described by some regular expression.
4. A language is regular if and only if it is generated by a left-regular grammar.
5. A language is regular if and only if it is generated by a right-regular grammar.

We will get to the definitions of left-regular and right-regular grammars later.

The Pumping Lemma

We can prove that a given language is regular by exhibiting a finite automaton which accepts it. The pumping lemma gives a technique for proving that certain languages are not regular.

The method is to first prove the pumping lemma, which states that every string w which is a member of some regular language L has a "pumpable" substring, namely a substring which can be duplicated without leaving L. We give the formal statement below.

We can then, for example, prove that $L = \{a^n b^n\}$ is not regular, by showing that there are arbitrarily long strings of L that do not have pumpable substrings.

Theorem 2 (Pumping Lemma) *For any regular language L, there is an integer p such that for any $w \in L$ of length at least p, there are strings x, y, z such that the following four conditions hold: Condition 1. $w = xyz$*

Condition 2. $|xy| \leq p$

Condition 3. y is not the empty string

Condition 4. For any integer $i \geq 0$, $xy^iz \in L$.

The number p is called a *pumping length* of L.

Proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA which accepts L, and let $p = |Q|$, the number of states of M.

Let $w \in L$ of length n, where $n \geq p$. Let a_i be the i^{th} symbol of w, that is, $w = a_1a_2 \cdots a_n$. Pick an accepting computation of M with input w. For $0 \leq t \leq n$, let $q^t \in Q$ be the state of M after t steps of that computation, that is, the state of M after reading $w[1 \ldots t] = a_1, \ldots a_t$. Note that $q^0 = q_0$, the start state of M, that $\delta(q_{t-1}, a_t) = q_t$ for all t, and that $q^n \in F$.

The set of states Q has size p, and the sequence of states $q^0, q^1, \cdots q^p$ has length $p + 1$ hence, by the pigeonhold principle,[3] the first p terms of the sequence must contain a duplicate; that is, $q^j = q^k$ for some $0 \leq j < k \leq p$. Thus, the computation path through M with input w has a loop, as shown in Figure 16. When that loop is excised, the resulting computation is still accepting, as shown in Figure 17. A computation which traverses the loop multiple times, as shown in Figure 18, is also accepting. We now define the strings x, y, and z. Let $x = w_{[1,j]} = a_1 \cdots a_j$, $y = w_{[]}j + 1, k] = a_{j+1} \cdots a_k$, and $z = w_{[]}k + 1, n] = a_{k+1} \cdots a_n$. We verify the four conditions of the pumping lemma. $xyz = a_1 \cdots a_n = w$,

[3]If each pigeon is in a pigeonhole and there are more pigeons than holes, at least one pigeonhole has at least two pigeons.

satisfying condition 1. $|xy| = k \leq p$, satisfying condition 2. $|y| = k - j >= 1$, satisfying condition 3. We need to prove condition 4, that is, that xy^iz is accepted by M for all $i \geq 0$. For $i = 0$, xz is accepted by M since $\delta(q^j, y) = q^k = q^j$, as shown in Figure 17. For $i = 1$, $x^iz = w \in L$. For $i > 1$, we have $\delta(q^j, y^i) = q^k = q^j$, hence M accepts xy^iz, as shown in Figure 18. ▮

In Figures 16, 17, and 18, $n = 10$, $j = 3$, and $k = 8$. to avoid clutter, we label each state q^t as simply t in the figures.

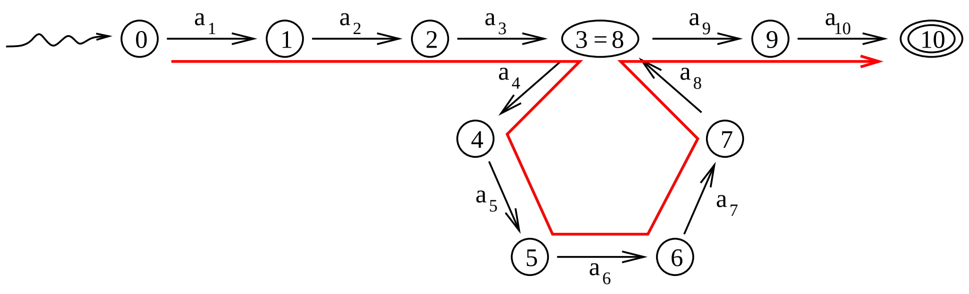

Figure 16: Computation of M with Input $w = xyz$

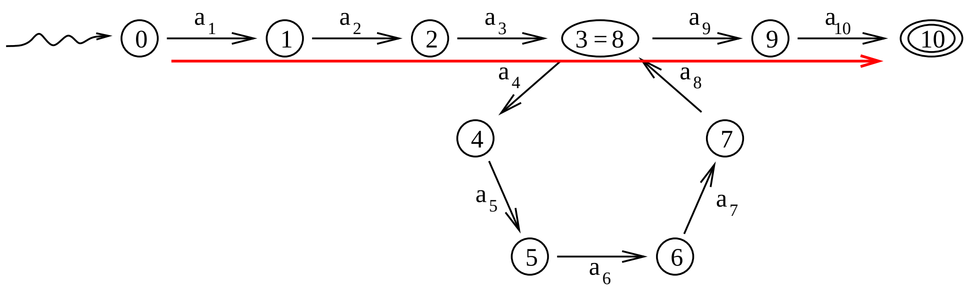

Figure 17: Computation of M with Input xz

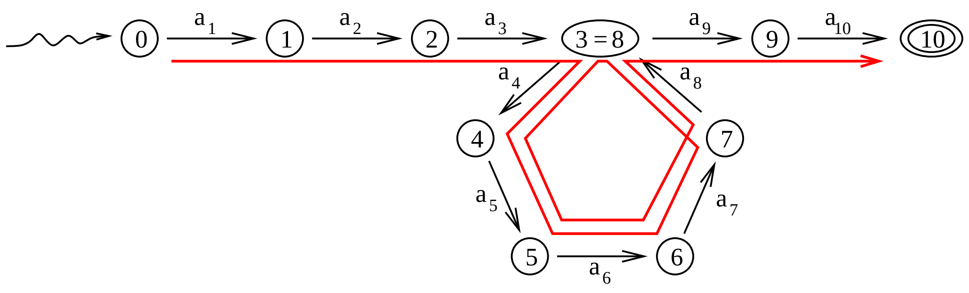

Figure 18: Computation of M with Input xy^2z

A Proof of Non-Regularity using the Pumping Lemma

We can use the pumping lemma to prove certain languages not to be regular, by contradiction.

Theorem 3 *Let $L = \{a^n b^n : n \geq 0\}$. Then L is not regular.*

Proof: By contradiction. Suppose L is regular. Let p be a pumping length of L. Let $w = a^p b^p \in L$. Note that $|w| \geq p$, hence there exist strings x, y, z which satisfy the four conditions of the pumping lemma. By Condition 1., $xyz = w$. Thus xy is a prefix of w. By Condition 2., $|xy| \leq p$, hence $xy = a^k$ for some $k \leq p$. By Condition 3., $y = a^l$ for $1 \leq \ell \leq k$. It follows that $x = a^{k-\ell}$ and $z = a^{p-k} b^p$. Thus $xz = a^{p-\ell} b^p$. By Condition 4., we can pick $i = 0$ and we then have $xy^0 z = xz \in L$. Since $\ell \geq 1$, xz has more b's than a's, and hence cannot be a member of L. Contradiction. We conclude that L is not regular. ▮