Formal Languages and Automata

Contents

1 Introduction

Welcome to Formal Languages and Automata, taught at UNLV as CS456/656. Every university which grants a degree in computer science should have an equivalent course. I expect every student to know high school mathematics as well as the material of our prerequisite courses, including programming and discrete mathematics. Our textbook is *Formal Languages and Automata*, by Peter Linz.

1.1 Formal Languages

1.1.1 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 (caveman) alphabet: $\{1\}$.
Small subsets of the Roman alphabet, such as $\{a, b\}$.

1.1.2 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.

1.1.3 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.

1.1.4 Programs

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. Pascal and C++ are examples of programming languages.

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

Natural Languages. In order for a set of strings to be a formal language, there must be a rigorous *mathematical* definition of that set of srrings. Natural languages, such as English, lack that definition, and are not studied in this course.

1.1.5 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.

1.2 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 example, the language for the primality problem is the set of all numerals for prime numbers. In general, if P is any 0,1 problem, the corresponding language is the set of all true instances of P. Convrsely, every language L over an alphabet Σ has a corresponding 0,1 problem, namely the membership problem for L. An instance is any string w over Σ, and the question is whether $w \in L$.

1.3 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 **id** 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 everyihing needed for the computation, such as the machine's current state, contents of its memory, unread input, and (possibly) written output. A string is necessarily finite, but during an infinite computation, the **id** could increase its length without limit.

1.3.1 Deterministic and Non-deterministic Machines

A machine *automaton*, plural *automata*, is called *deterministic* if for a given **id** and a given input, there is at most one step the machine can take, *i.e.,* at most one immediately subsequent id. A machine is called *non-deterministic* there there is some number of choices at each step. Note that "some number" could be always 0 or 1, and thus every deterministic machine is a non-deterministic machine

1.3.2 Accept and Decide

We say that a non-deterministic machine M *accepts* a string w if, given any input w, M halts in an *accepting* state. We say M *accepts* a 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 $w \notin L$. It is possible for M to run forever with some input w, neither accepting nor rejecting. In that case, we say that M does not accept w, although we might never know it.

1.3.3 Deterministic Finite Automata

A machine M is called a *finite automaton* (FA) if it has finitely many states. A deterministic FA is called a DFA, while a non-deterministic FA is called an NFA. As noted above, every DFA is an NFA, but not vice-versa. A DFA always decides a language, but most textbooks refer to the language *accepted* by a DFA M, which is (of course) the language decided by M.

1.3.4 Computation of a DFA

A 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. The definition of DFA requires that there be *exactly* one action that a DFA can take, and that action is either to change state or to halt.

Formally, M is the quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states of M, Σ is the alphabet of M, $\delta : Q \times \Sigma \to Q$ is the *transition function* of M, $q_0 \in Q$ is the *start state* of M and $F \subseteq Q$ is the set of *final states* of M.

An **id** of a DFA 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. The steps of a computation of M are as follows. Suppose the current id is (q, w).

1. If w is the empty string M halts and accepts if $q \in F$, halts and rejects if $q \notin F$.
2. Otherwise, w is the concatenation aw'.
3. M reads a, and w' becomes the remaining input.
4. M changes its state to $\delta(q, a)$.

The number of steps a DFA executes during a computation equals the length of the input string.

Regular Languages. A *regular* language is any language accepted by a DFA.

Example. Let M_1 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 **??**, and illustrated as a state diagram in Figure **??**

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

Table ??

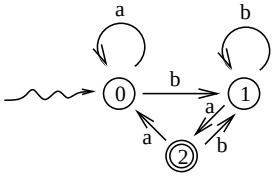

Figure ??: State Diagram of M

Figure **??** shows a computation of M_1 which accepts the string *abba*, while Figure **??** shows a computation of M_1 which rejects the string *abab*.

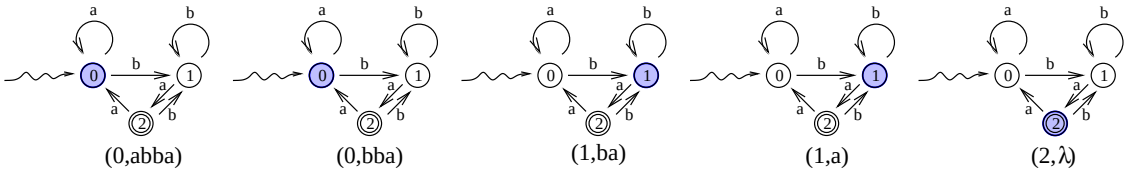

Figure ??: Computation of M_1 accepting *abba*. For simplicity, the states are labeled 0, 1, 2 instead of q_0, q_1, q_2. The final state is doubly circled. The figures show the sequence of **ids**. The current state is indicated in blue, and the current **id** is underneath the figure. Note that the last state is final.

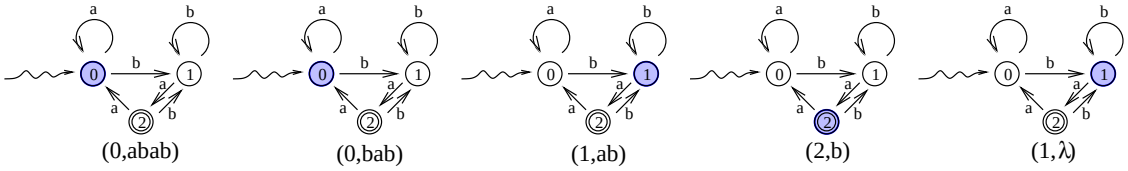

Figure ??: Computation of M_1 rejecting *abab*. Note that the last state is not final.

Let $L = L(M_1)$ be the language accepted by M_1. Name three members of L and three strings which are not members of L. How would you describe L?

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 input. The machine shown above, in Figures **??** and **??**, does not have a dead state. The machine shown in Figure **??** has a dead state, q_1.

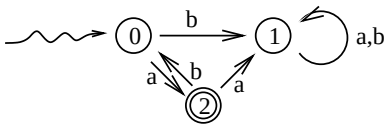

Figure ??: A DFA M_2 with one dead state. Note that it is impossible to leave state q_1.

How would you describe the language $L(M_2)$?

1.3.5 Non-Deterministic Finite Automata

An NFA has the same basic structure as a DFA, but for a given state and input symbol, there are any number of subsequent states.

For any set S, we write 2^S, for the *powerset* of S,

In a computation of an NFA, if the current state is q and the next input is a, then the machine may move to any member of the set $\Delta(q, a) \subseteq Q$. 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 the transition function is $\Delta : Q \times \Sigma \cup \{\lambda\} \to 2^Q$.

1.3.6 Example.

Let M_3 be the NFA whose state diagram is shown in Figure **??**.

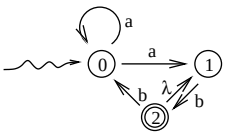

Figure ??: NFA M_3

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 ??: Transition Table of M_3

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_3 with input *abb* in Figure **??**. At the first step, M_3 reads a and moves to q_1. Alternatively, M_3 could read a and stay in q_0, but then it would be impossible to accept the input. When we are analyzing whether an NFA accepts a string, in case of a choice, we assume that the NFA always make a choice which leads to acceptance, if that is possible.

At the third step, M_3 has another choice. M_3 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.

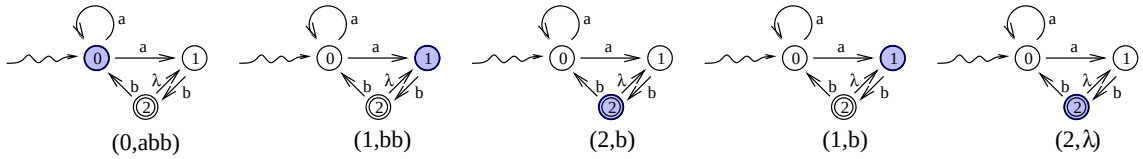

Figure ??: Accepting Computation of M_3 with Input *abb*.

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 **??**.

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 minimizing a DFA. Let M be a DFA which accepts a language L over an alphabet Σ. Hopcroft's 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. Let M_4 be the DFA illustrated by the state diagram in Figure **??**.

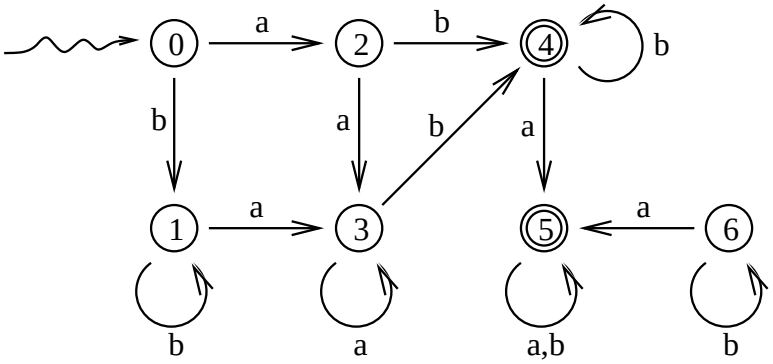

Figure ??: The DFA M

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 $Q \times \Sigma \times Q$. AAA 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			×		×	×
q_1					×	×
q_2	×				×	×
q_3					×	×
q_4	×	×	×	×		
q_5	×	×	×	×		

BBB 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	×	×	×	×		

CCC We iterate over $Q \times \Sigma \times Q$ until no additional 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 "collapsed" DFA M_5 is equivalent to M_4, and is minimal. We illustrate M_5 in Figure **??**.

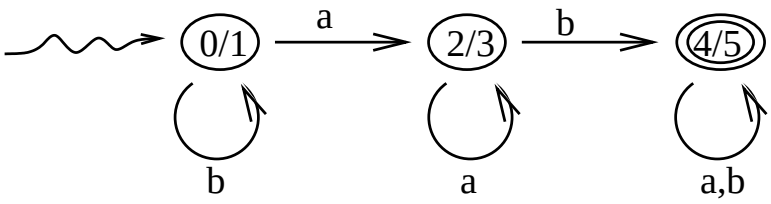

Figure ??: M_5, the Minimal DFA Equivalent to M_4.

Another Example. Let M_6) be the DFA illustrated by the state diagram in Figure **??**.

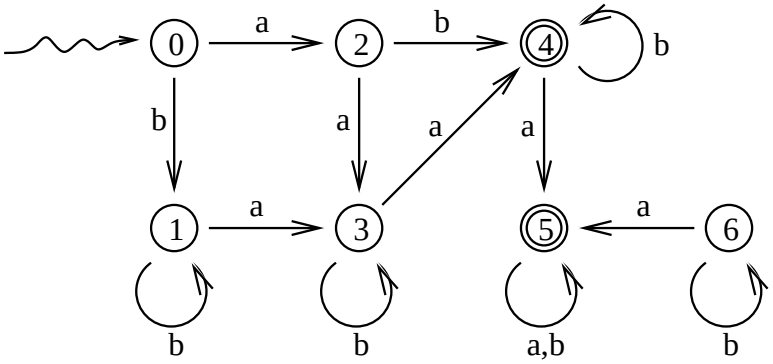

Figure ??: The DFA M_6

We first note that q_6 is useless, hence we delete it. As in the previous example, We write a square array whose rows and columns are the remaining states. 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 $Q \times \Sigma \times Q$. During the first iteration, we observe q_2 and q_3 are distinguished. because $\delta(q_2, a) = q_3$ and $\delta(q_3, a) = q_4$ are distinguished.

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

During the second iteration, we 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	×	×	×	×		

III 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 pairs (q_4, q_5). The resulting minimal DFA equivalent to M_6 is illustrated in Figure **??**.

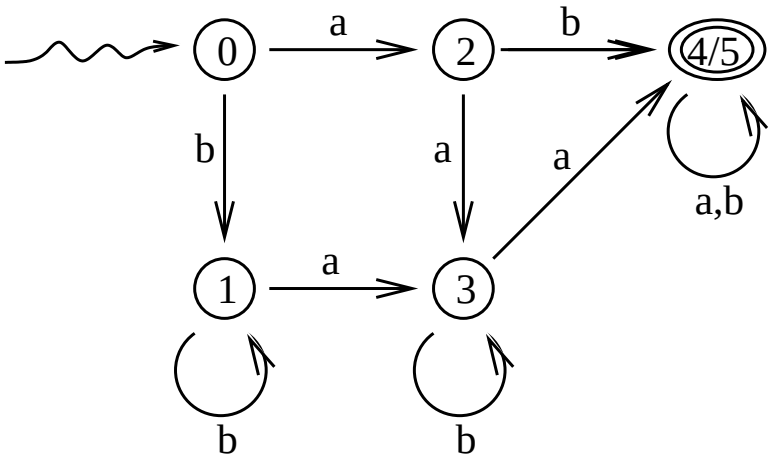

Figure ??: The minimal DFA equivalent to M_6.

NFA to DFA

Two finite automat are *equivalent* if they accept the same language. A language is regular if and only if it is accepted by some NFA, and thus every NFA is equivalent to some 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, 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 **??** shows the complete calculation of a minimal DFA equivalent to an NFA.

Figure **??**(a) shows the NFA M_1.

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

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

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

Figure **??**(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 **??**(f).

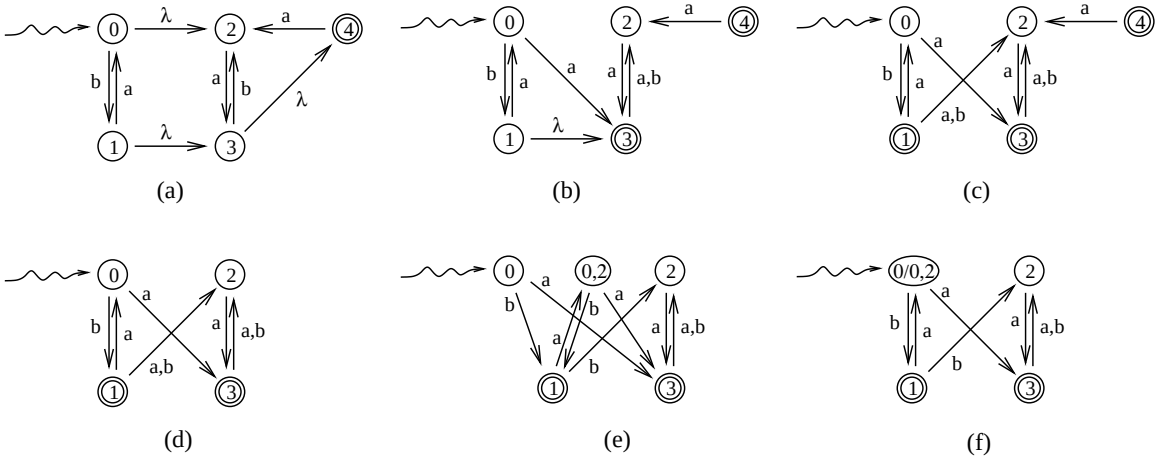

Figure ??: Construction of a Minimal DFA Equivalent to an NFA.

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 **??** whose state diagram is given by Figure **??**. meaning that every final state becomes non-final and every non-final state becomes final. (This trick does not work for an NFA.)

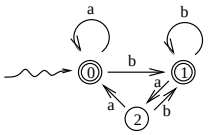

Figure ??: State Diagram of a DFA which Accepts the Complement of the Language Accepted by the DFA whose State Diagram is Shown in Figure **??**.

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 be the alphabet

of L, a regular expression for L is a string over the alphabet $\Sigma + \{\lambda, \emptyset, +, ^*, (,)\}$.

Regular expressions for languages over Σ are algebraic expressions, where the variables are the symbols of Σ, together with λ and $\emptyset$; 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 $\emptyset$ represents the empty language, $\emptyset$.

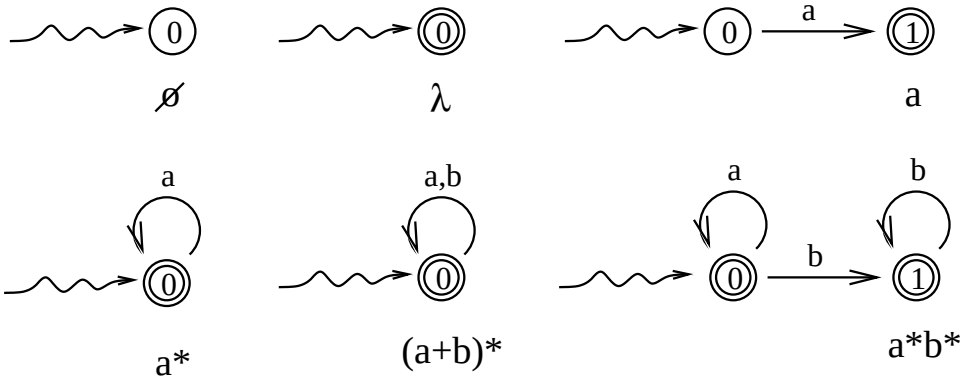

Figure ??: Regular Expressions over $\Sigma = \{a, b\}$ with Equivalent DFA.

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 **??** 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 $i \geq 0$ $xy^i z \in L$.*

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.

Let $w \in L$ of length n, where $n \geq p$. Let a_i be the i^{th} symbol of w, that is, $w = a_1 a_2 \cdots a_n$. 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 For each $0 \leq t \leq n$, Let $w_{[1,t]} = w_1 w_2 \cdots w_t$, the prefix of w of length t. For each $0 \leq t \leq n$ let q^t be the state of M after t steps of the computation of M with input w. Note that $q^0 = q_0$, the start state of M, and that $\delta(q_{t-1}, a_t) = q_t$ for all t. Thus $\delta(q_0, w) = q^n \in F$.

The set of states Q has size p, and the sequence of states $q^0, q^1, \cdots q^p$ has length $p+1$. The sequence must then 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 **??**. When that loop is excised, the resulting computation is still accepting, as shown in Figure **??**. A computation which traverses the loop multiple times, as shown in Figure **??**, is also accepting. We use those observations to prove condition 4. below.

We now define the strings x, y, and z. Let $x = w_{[1,j]} = a_1 \cdots a_j$. Let $y = a_{j+1} \cdots a_k$, and let $z = a_{k+1} \cdots a_n$. Thus $xyz = a_1 \cdots a_n = w$, satisfying condition 1. $|xy| = k \leq p$, satisfying condition 2. $|y| = k - j >= 1$, satisfying condition 3.

We need to prove condition 4. Let $i \geq 0$; we will show that $xy^i z$ is accepted by M. We have $\delta(q^j, y) = q^k = q^j$. Repeating y, we have $\delta(q^j, y^i) = q^k = q^j$. Finally, $\delta(q^0, xy^i z) = q^n \in F$, hence $xy^i z \in L$. ∎

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

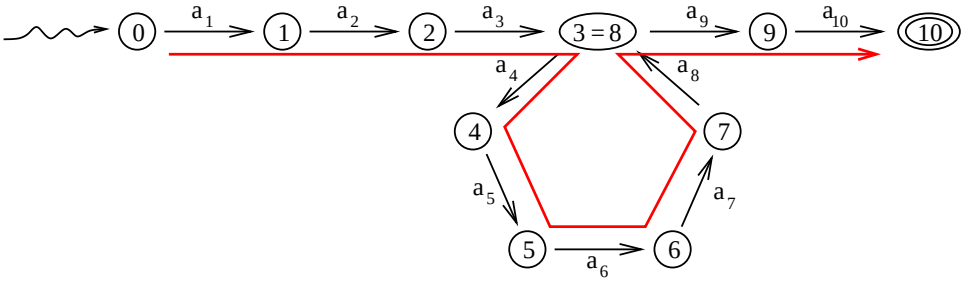

Figure ??: Computation of M with Input $w = xyz$

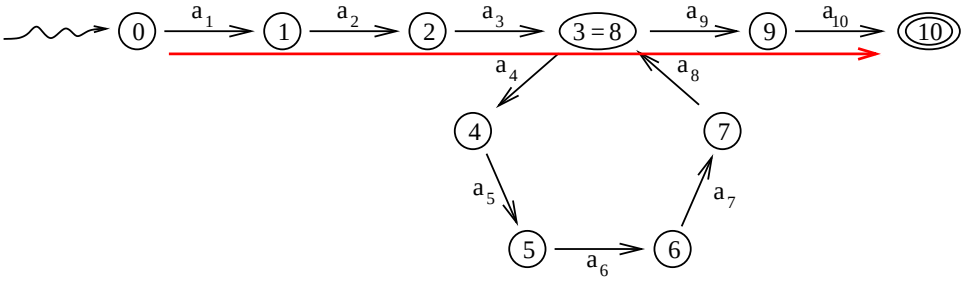

Figure ??: Computation of M with Input xz

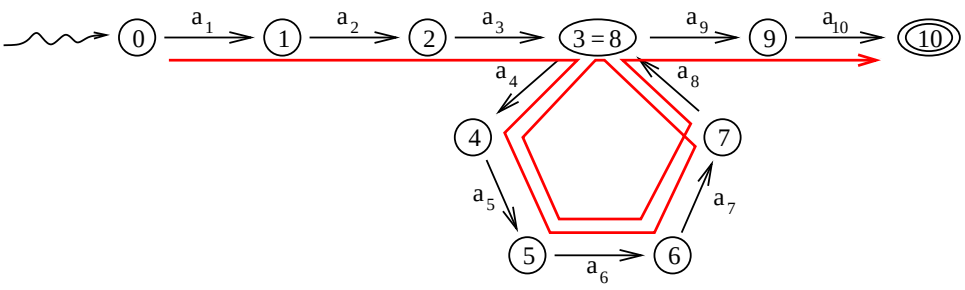

Figure ??: Computation of M with Input xy^2z

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 $w = a^p b^p$. 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 $xy^0z = xz \in L$. 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. ∎