1 Introduction

Welcome to Formal Languages and Automata, taught at UNLV as CS456/656. 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.

2 Formal Languages

A language is a set of strings (also called words) of symbols over a given alphabet. Natural languages, such as English, are not studied in this course; the problem is that every person speaks a different version, and that a person's version is constantly changing: you do not speak the same language that you did at age five.

A simple example of a formal language that everyone knows is the set of decimal (base 10) numerals for positive integers. A more complex example is any programming language, such as C++. From now on, when we say "language" we mean "formal language." Whenever We ask a class, on the first day, what a language is, the word "communication" is inevitably mentioned. But that is not part of the definition. We give the formal definition of a "language" below.

Symbols, Every kind of discussion must begin with undefined terms, such as the term "point" in plane geometry. The word "symbol" is an undefined term. That does not imply that there's no such thing as a symbol: in fact, anything could be called a symbol.

Alphabets. An alphabet is a finite set of symbols. Here are some alphabets in common use.

1. The Roman alphabet, either lower or upper case, or both.
2. The alphabet of Arabic digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
3. The alphabet of all ASCII symbols.
4. The binary alphabet: $\{0, 1\}$.
5. The unary (caveman) alphabet: $\{1\}$. (Is it still in use today?)

Strings. A string is a finite sequence of symbols, such as *abbaba* or 1024. If w is a string, we write $|w|$ for the length of w. For example, $|abbaba| = 6$. The *empty* string is the string of length zero. We will use the Greek letter λ to denote the empty string, but most textbooks use the Greek letter ϵ instead.

Languages Given an alphabet Σ, a language over Σ is a set of strings over Σ, *i.e.,* a string of symbols of Σ. A language can be finite or infinite. Here are some examples.

1. Σ^*, The set of all strings over a given alphabet Σ.
2. $\{\lambda\}$, the language consising of just one string, the empty string.

3. $\emptyset$, the empty set, which we also call the empty language.

4. The set of all decimal (base 10) numerals for positive integers.

5. The set of all words used by Shakespeare.

6. The set of all palindromes over the alphabet $\{a, b\}$, such as a, aa, $abba$, $abbababba$.

7. C++, the set of all C++ programs.

2.1 Operations on Languages

A language is a set, and the standard operations on sets apply to languages as well. These include union intersection, inclusion, and complementation. It is common to use the symbol ''+" to denote a union of languages. In addition, we have the operations *concatenation* and *Kleene closure*.

If A and B are languages, the concatenation AB is the set of all strings consisting of the concatenation of a member of A and a member of B. For example, if $A = \{a, ab\}$ and $B = \{a, ba\}$, then $AB = \{aa, aba, abba\}$. We write A^2 for the concatenation AA. In our example, $A^2 = \{aa, aba, abab\}$ and $A^3 = A^2A$. Concatenation is associative but not commutative. That is, for any languages A, B, C, $(AB)C = A(BC)$, but BA may not equal AB.

If L is any language, the *Kleene closure* of L, written L^*, is the set of all strings, which are concatentations of any sequence of members of L. We also write $L^* = L^0 + L^1 + L^2 + L^3 + \cdots$. For example, if $L = \{a\}$, $L^* = \{\lambda, a, aa, aaa, \ldots\}$. For any language L, the empty string is a member of L^*.

Exercises. Let A be any language over an alphabet Σ, and $a, b \in \Sigma$.

1. What is $A + A$?

2. What is $\{\lambda\}A$? (Concatenation)

3. What is $\{a, ab\}\{a, ba\}$?

4. What is $\emptyset A$?

5. What is $\{\lambda\}^*$?

6. What is $\emptyset^*$?

7. Is it true that $\{\lambda\} + A = A$?

3 Languages and Machines

An abstract *machine*, or *automaton*,[1], is a mathematical object. We do not mean a physical machine such as your laptop or your car. A *computation* of a machine M consists of discrete *steps*, each of which takes finite time. ("Time" is measured in abstract "time units," rather than in physical time units such as seconds.) At the beginning of any computation, M is in its *start state*. At each step, M executes some combination of actions, such as:

[1] https://en. wikipedia.org/wiki/Automaton

1. M can from its *input*.
2. M can change its state.
3. M can write to its *output*.
4. M can fetch from, and store to, its memory.
5. M can halt.

We say a machine M is *deterministic* if there is never more than one step M can execute. If M has not halted and has no possible step, we say M *hangs*. We will study a number of classes of machines. The simplest such class is *deterministic finite automata*.

A DFA, deterministic finite automaton, is defined to be a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set, the set of states of M
2. Σ is the alphabet of M. All inputs of M must be strings over Σ.
3. $\delta : Q \times \Sigma \to Q$ is the *transition* function of M.
4. $q_0 \in Q$ is the start state of M.
5. $F \subseteq Q$ is the set of *final states* of M.
6. A DFA knows its current state, but has no additional memory.

3.1 Regular Languages

A language L is *regular* if it is *accepted* by some DFA. Let M be the DFA $(Q, \Sigma, \delta, q_0, F)$. Let $w \in \Sigma^*$. With input string w, a computation of M is as follows.

Initially M is in the start state q_0.
While there is still unread input
  Let q be the current state of M, and a the next unread input symbol.
  M reads a.
  M changes state to $\delta(a, q)$.
If the current state of M is final M reports acceptance.
Else M reports rejection.

$L(M)$, the language accepted by M, is the set of all strings over Σ accepted by M.

3.1.1 State Diagrams

A DFA M $(Q, \Sigma, \delta, q_0, F)$ is represented by a *state diagraph*[2] is a labeled directed graph, which is conventionally drawn as a labeled directed graph.

[2] https://en.wikipedia.org/wiki/ Deterministic_finite_automaton

Q is the set of vertices of the diagraph.. The diagram includes an arrow (sometimes drawn with wiggles) pointing at the start state, q_0.

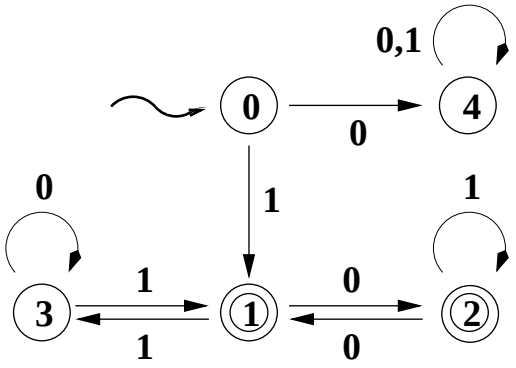
It is standard practice to show each non-final state as a labeled circle, and each final state as a labeled double circle. To decrease business of the figure, I typically label the state q_i with just the integer i.

A *transition* is an ordered triple (q_i, a, q_j) such that $\delta(a_i, a) = q_j$, conventionally llustrated by a arc labeled a from q_i to q_j. If there is more than one transition from q_i to q_j, typically only one arc with multiple labels is shown.

M can also be represented as a matrix, where each row is label by a state and each column by a member of Σ.

Example: Numerals 1 or 2 Modulo 3 L consists of all binary numerals for positive numbers not divisible by 3, where leading zeros are not allowed. That is, $L = \{1, 10, 100, 101, 111, 1000, 1010, \ldots\}$ Below we show the transition matrix and the state diagram of a DFA M which accepts L. M has five states and two final states, q_1 and q_2.

	0	1
q_0	q_4	q_1
q_1	q_2	q_3
q_2	q_1	q_2
q_3	q_3	q_1
q_4	q_4	q_4



Dead States. A DFA can have a *dead state*, namely a non-final state from which no computation can "escape." That means, if a computation reaches a dead state, then the machine will not accept regardless of the rest of the input string. The DFA shown above has a dead state, namely q_4.

3.1.2 Deterministic and Non-deterministic Machines

The *instaneous description*, **id**, of a machine is a string which encodes all significant data of the machine at a given instant. At each step of a computation, the **id** is updated. A machine is called *deterministic* if for a given **id** and a given input, there is at most at most one immediately subsequent **id**, that is, the **id** after the next step. A machine is called *non-deterministic* if there are some number of choices of **id** for the next step. Note that "some number" could be always 0 or 1, hence a deterministic machine is also a non-deterministic machine.

A *non-deterministic automaton*, abbreviated NFA, is almost the same as a DFA. An NFA is a quintuple $M = (Q, \Delta, \Sigma, q_0, F)$ such that

1. Q is the finite set of states of M

2. Σ is the alphabet of M.

3. $\Delta : Q \times \Sigma \to 2^Q$ is the *transition* function of M. (Recall that 2^S, the *powerset* of a set S, is the set of subsets of S.) That is, if the current state of M is $q \in Q$ and M reads the symbol $a \in \Sigma$, M can change state to any member of the set $\Delta(q, a)$.

4. $q_0 \in Q$ is the start state of M.

5. $F \subseteq Q$ is the set of *final states* of M.

A computation of an NFA is similar to a computation of a DFA. At each step M reads a symbol and changes state, although there might not be a single choice of that new state.

3.1.3 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.

The definition of acceptance by a non-deterministic machine is trickier. For a given input string w, there could be some computation of M which ends at a final state, but that computation might depend on M making correct choices. If there are computations of M with input w that end in a final state, we say that M *accepts* w, despite the possibility that some other sequence of choices does not end in a final state. We call this *benevolent non-determinism* because we assume M makes all the correct guesses. The class of languages accepted by non-deterministic automata is the regular languages.

Example Let M_1 be the DFA whose transition matrix and state diagram are shown below.

	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_1

Transition Table of M_1

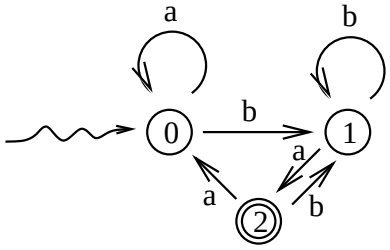


Figure 1: State Diagram of M_1

Figure 2 shows a computation of M_1 which accepts the string *abba*, while Figure 3 shows a computation of M_1 which rejects the string *abab*.

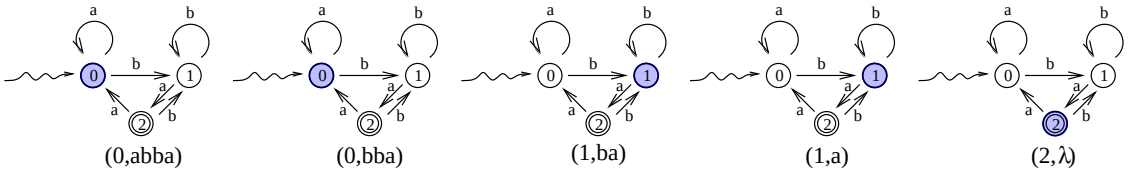


Figure 2: Computation of M_1 accepting *abba*. For simplicity, the states are labeled 0, 1, 2 instead of q_0, q_1, q_2. The final state is doubly circled. The figures show the sequence of **ids**. The current state is indicated in blue, and the current **id** is underneath the figure. Note that the last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

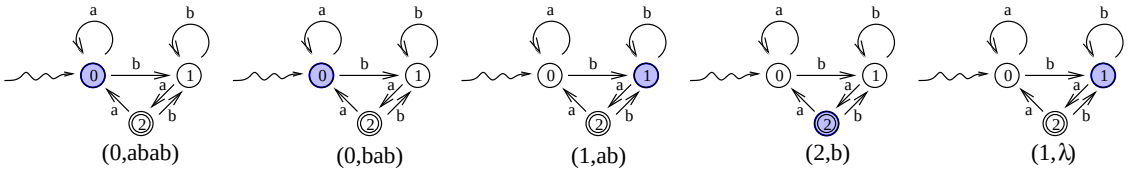


Figure 3: Computation of M_1 rejecting *abab*. Note that the last state is not final.

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?

3.1.4 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 $\Delta : Q \times \Sigma \cup \{\lambda\} \to 2^Q$.

3.1.5 Example.

Let M_3 be the NFA whose state diagram is shown in Figure 4.

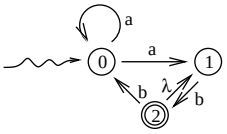


Figure 4: NFA M_3

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 5: 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 6. 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.

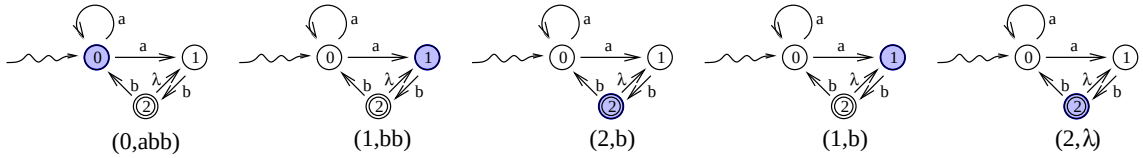


Figure 6: Accepting Computation of M_3 with Input *abb*.

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).

3.2 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.

3.2.1 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 7.

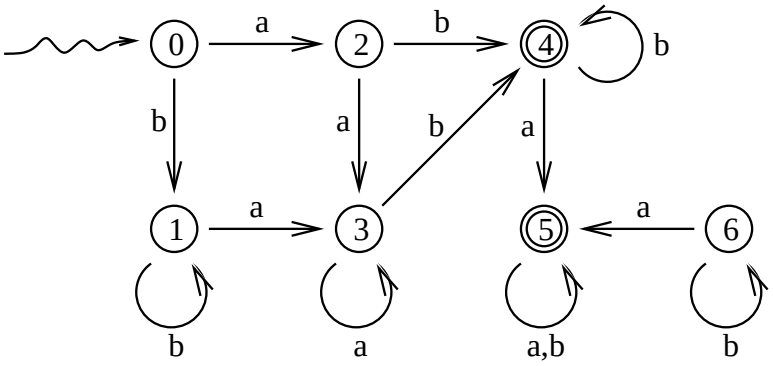


Figure 7: The DFA M

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 8.

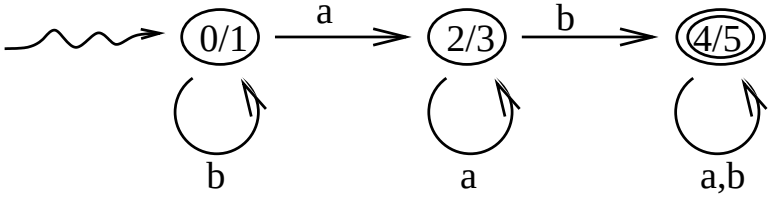


Figure 8: M_5, the Minimal DFA Equivalent to M_4.

Another Example. Let M_6) be the DFA illustrated by the state diagram in Figure 9.

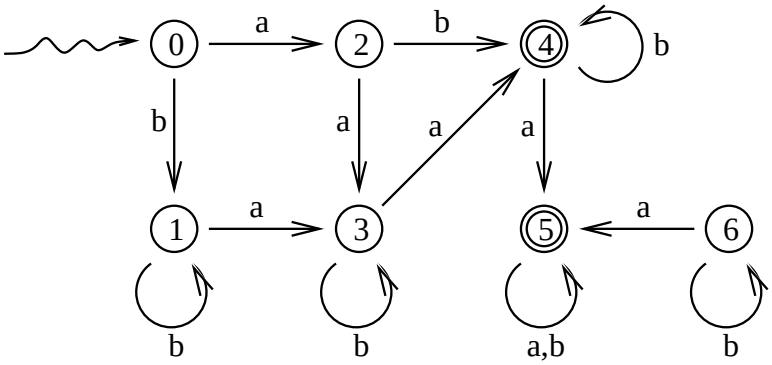


Figure 9: The DFA M_6

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 10.

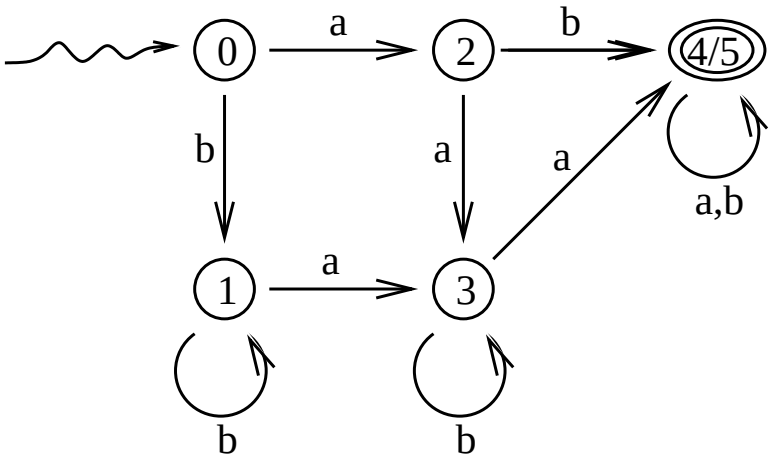


Figure 10: The minimal DFA equivalent to M_6.

3.3 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 11 shows the complete calculation of a minimal DFA equivalent to an NFA.

Figure 11(a) shows the NFA M_1.

Figure 11(b) shows the NFA after the λ-transitions from q_0 to q_2 and from q_3 to q_4 are removed.

Figure 11(c) shows the NFA after the λ-transition from q_1 to q_3 is removed, yielding the λ-closure of M_2.

Figure 11(d) is obtained by deleting the now useless state q_4.

Figure 11(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 11(f).

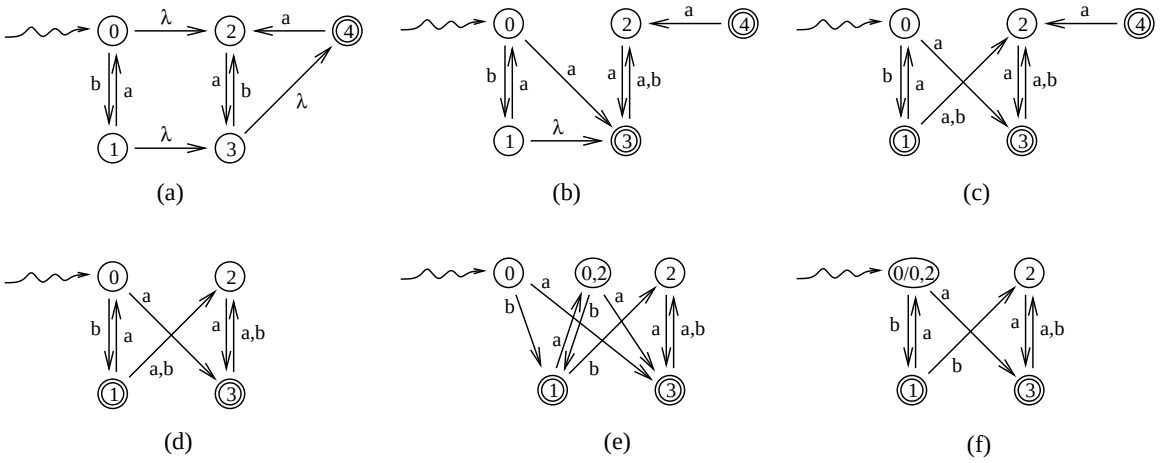


Figure 11: Construction of a Minimal DFA Equivalent to an NFA.

4 Regular Expressions

A regular expression is an algebraic expression that defines, or describes, a regular language. Regular expressions make use of three of the *closure properties* of the class of regular languages given in Section 2.1: union, concatenation, and Kleene closure.

Every regular language is described by a regular expression. Regular expressions which describe the same language are called *equivalent*. If Σ be the alphabet of L, a regular expression for L is a string over the alphabet $\Sigma + \{\lambda, \emptyset, +, ^*, (,)\}$. concatenation does not have an operator symbol, but is simply indicated by juxtaposition. For example, the regular expressions a and b describe the languages $\{a\}$ and $\{b\}$, respectively, and thus ab is a regular expression which describes $\{ab\}$. Union is represented by "+", while Kleene Closure is reprented by $*$. Kleene closure has precedence over concatenation, which has precedence over union. Parentheses are used in the usual way, to override precedence.

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, $\emptyset$.

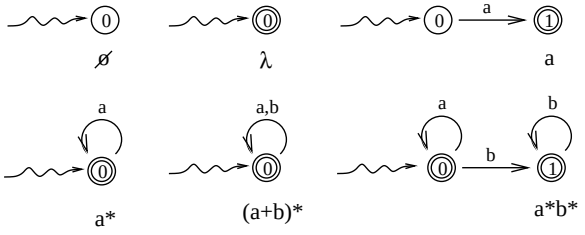


Figure 12: Regular Expressions over $\Sigma = \{a, b\}$ with Equivalent DFA.

4.1 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 12 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 give the definitions of left-regular and right-regular grammars later.

5 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 13. When that loop is excised, the resulting computation is still accepting, as shown in Figure 14. A computation which traverses the loop multiple times, as shown in Figure 15, 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 13, 14, and 15, $n = 10$, $j = 3$, and $k = 8$. to avoid clutter, we label each state q^t as simply t in the figures.

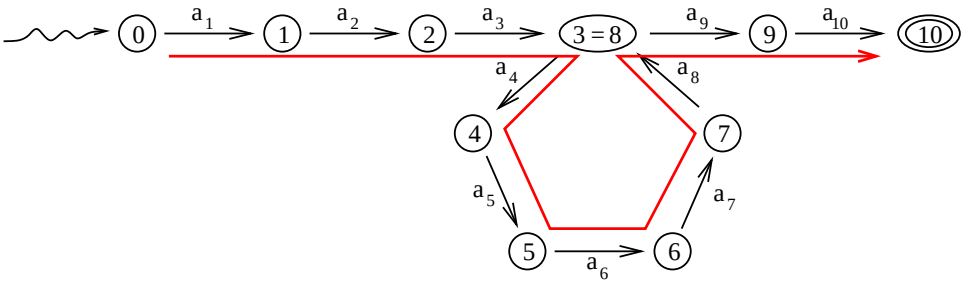


Figure 13: Computation of M with Input $w = xyz$

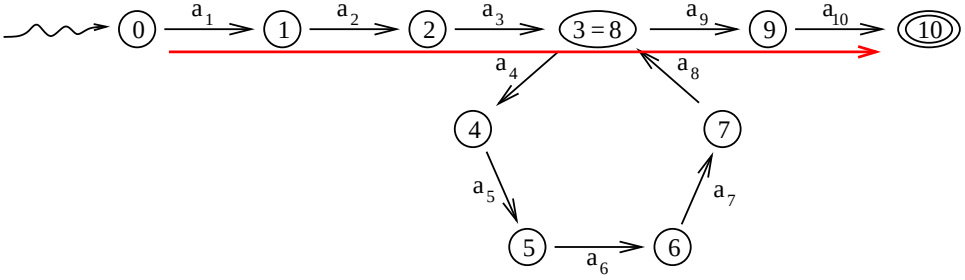


Figure 14: Computation of M with Input xz

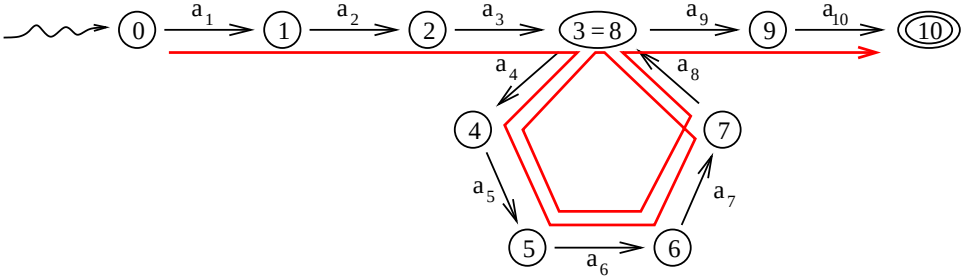


Figure 15: Computation of M with Input xy^2z

5.0.1 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. ∎