Regular Languages and Finite State Machines

1 Regular Languages

The first class of languages that we will consider is the regular languages. As we'll see later, there are several quite different ways in which regular languages can be defined, but we'll start with one simple technique, regular expressions.

A ***regular expression*** is an expression (string) whose "value" is a language. Just as 3 + 4 and 14/2 are two arithmetic expressions whose values are equal, so are $(a \cup b)^*$ and $a^* \cup (a \cup b)^*b(a \cup b)^*$ two regular expressions whose values are equal. We will use regular expressions to denote languages just as we use arithmetic expressions to denote numbers. Just as there are some numbers, like Π, that cannot be expressed by arithmetic expressions of integers, so too there are some languages, like a^nb^n, that cannot be expressed by regular expressions. In fact, we will define the class of ***regular languages*** to be precisely those that *can* be described with regular expressions.

Let's continue with the analogy between arithmetic expressions and regular expressions. The syntax of arithmetic expressions is defined recursively:

1. Any numeral in {0, 1,2, ...} is an arithmetic expression.
2. If α and β are expressions, so is $(\alpha + \beta)$.
3. If α and β are expressions, so is $(\alpha * \beta)$.
4. If α and β are expressions, so is $(\alpha - \beta)$.
5. If α and β are expressions, so is (α/β).
6. If α is an expression, so is $-\alpha$.

These operators that we've just defined have associated with them a set of precedence rules, so we can write -3 + 4*5 instead of (-3 + (4*5)).

Now let's return to regular expressions. The syntax of regular expressions is also defined recursively:

1. $\varnothing$ and each member of Σ is a regular expression.
2. If α , β are regular expressions, then so is $\alpha\beta$
3. If α , β are regular expressions, then so is $\alpha \cup \beta$.
4. If α is a regular expression, then so is α^*.
5. If α is a regular expression, then so is (α).
6. Nothing else is a regular expression.

Similarly there are precedence rules for regular expressions, so we can write $a^* \cup bc$ instead of $(a^* \cup (bc))$. Note that * binds more tightly than does concatenation, which binds more tightly than $\cup$.

In both cases (arithmetic expressions and regular expressions) there is a distinction between the expression and its value. 5 + 7 is not the same expression as 3 * 4, even though they both have the same value. (You might imagine the expressions being in quotation marks: "5 + 7" is clearly not the same as "3 * 4". Similarly, "John's a good guy" is a different sentence from "John is nice", even though they both have the same meaning, more or less.) The rules that determine the value of an arithmetic expression are quite familiar to us, so much so that we are usually not consciously aware of them. But regular expressions are less familiar, so we will explicitly specify the rules that define the values of regular expressions. We do this by recursion on the structure of the expression. Just remember that the regular expression itself is a syntactic object made of parentheses and other symbols, whereas its value is a language. We define L() to be the function that maps regular expressions to their values. We might analogously define L() for arithmetic expressions and say that L(5 + 7) = 12 = L(3 * 4). L() is an example of a ***semantic interpretation function***, i. e., it is a function that maps a string in some language to its meaning. In our case, of course, the language from which the arguments to L() will be drawn is the language of regular expressions as defined above.

L() for regular expressions is defined as follows:

1. $L(\varnothing) = \varnothing$ and $L(a) = \{a\}$ for each $a \in \Sigma$
2. If α, β are regular expressions, then

$L((\alpha\beta)) = L(\alpha)\ L(\beta)$

= all strings that can be formed by concatenating to some string from α some string from β.

Note that if either α or β is $\varnothing$, then its language is $\varnothing$, so there is nothing to concatenate and the result is $\varnothing$.

3. If α, β are regular expressions, then $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$
4. If α is a regular expression, then $L(\alpha^*) = L(\alpha)^*$
5. $L((\alpha)) = L(\alpha)$

So, for example, let's find the meaning of the regular expression $(a \cup b)^*b$:

$L((a \cup b)^*b)$
$= L((a \cup b)^*)\ L(b)$
$= L(a \cup b)^*\ L(b)$
$= (L(a) \cup L(b))^*\ L(b)$
$= (\{a\} \cup \{b\})^*\ \{b\}$
$= \{a, b\}^*\ \{b\}$

which is just the set of all strings ending in b. Another example is $L(((a \cup b)(a \cup b))a(a \cup b)^*) = \{xay: x \text{ and } y \text{ are strings of a's and b's and } |x| = 2\}$. The distinction between an expression and its meaning is somewhat pedantic, but you should try to understand it. We will usually not actually write L() because it is generally clear from context whether we mean the regular expression or the language denoted by it. For example, $a \in (a \cup b)^*$ is technically meaningless since $(a \cup b)^*$ is a regular expression, not a set. Nonetheless, we use it as a reasonable abbreviation for $a \in L((a \cup b)^*)$, just as we write 3 + 4 = 4 + 3 to mean that the values of "3 + 4" and "4 + 3", not the expressions themselves, are identical.

Here are some useful facts about regular expressions and the languages they describe:

- $(a \cup b)^* = (a^*b^*)^* = (b^*a^*)^*$ = set of all strings composed exclusively of a's and b's (including the empty string)
- $(a \cup b)c = (ac \cup bc)$ Concatenation distributes over union
- $c(a \cup b) = (ca \cup cb)$ "
- $a^* \cup b^* \neq (a \cup b)^*$ The right-hand expression denotes a set containing strings of mixed a's and b's, while the left-hand expression does not.
- $(ab)^* \neq a^*b^*$ In the right-hand expression, all a's must precede all b's. That's not true for the left-hand expression.
- $a^* \cup \varnothing^* = a^* \cup \varepsilon = a^*$

There is an algebra of regular expressions, but it is rather complex and not worth the effort to learn it. Therefore, we will rely primarily on our knowledge of what the expressions mean to determine the equivalence (or non-equivalence) or regular expressions.

We are now in a position to state formally our definition of the class of ***regular languages***: Given an alphabet Σ, the set of regular languages over Σ is precisely the set of languages that can be defined using regular expressions with respect to Σ. Another equivalent definition (given our definition of regular expressions and L()), is that the set of regular languages over an alphabet Σ is the smallest set that contains $\varnothing$ and each of the elements of Σ, and that is closed under the operations of concatenation, union, and Kleene star (rules 2, 3, and 4 above).

2 Proof of the Equivalence of Nondeterministic and Deterministic FSAs

In the lecture notes, we stated the following:

> ***Theorem***: For each NDFSA, there is an equivalent DFSA.

This is an extremely significant theorem. It says that, for finite state automata, nondeterminism doesn't add power. It adds convenience, but not power. As we'll see later, this is not true for all other classes of automata.

In the notes, we provided the first step of a proof of this theorem, namely an algorithm to construct, from any NDFSA, an equivalent DFSA. Recall that the algorithm we presented was the following: Given a nondeterministic FSA M $(K, \Sigma, \Delta, s, F)$, we derive an equivalent deterministic FSA $M' = (K', \Sigma, \delta', s', F')$ as follows:

1. Compute E(q) for each q in K. $\forall q \in K$, $E(q) = \{p \in K : (q, \varepsilon) \vdash^*_M (p, \varepsilon)\}$. In other words, E(q) is the set of states reachable from q without consuming any input.
2. Compute $s' = E(s)$.
3. Compute δ', which is defined as follows: $\forall Q \in 2^K$ and $\forall a \in \Sigma$, $\delta'(Q, a) = \cup\{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$. Recall that the elements of 2^K are sets of states from the original machine M. So what we've just said is that to compute the transition out of one of these "set" states, find all the transitions out of the component states in the original machine, then find all the states reachable from them via epsilon transitions. The new state is the set of all states reachable in this way. We'll actually compute δ' by first computing it for the new start state s' and each of the elements of Σ. Each state thus created becomes an element of K', the set of states of M'. Then we compute δ' for any new states that were just created. We continue until there are no additional reachable states. (so although δ' is defined for all possible subsets of K, we'll only bother to compute it for the reachable such subsets and in fact we'll define K' to include just the reachable configurations.)
4. Compute K' = that subset of 2^K that is reachable, via δ', as defined in step 3, from s'.
5. Compute $F' = \{Q \in K' : Q \cap F \neq \varnothing\}$. In other words, each constructed "set" state that contains at least one final state from the original machine M becomes a final state in M'.

However, to complete the proof of the theorem that asserts that there is an *equivalent* DFSA for every NDFSA, we need next to prove that the algorithm we have defined does in fact produce a machine that is (1) deterministic, and (2) equivalent to the original machine.

Proving (1) is trivial. By the definition in step 3 of δ', we are guaranteed that δ' is defined for all reachable elements of K' and that it is single valued.

Next we must prove (2). In other words, we must prove that M' accepts a string w if and only if M accepts w. We constructed the transition function δ' of M' so that each step of the operation of M' mimics an "all paths" simulation of M. So we believe that the two machines are identical, but how can we prove that they are? Suppose we could prove the following:

> ***Lemma***: For any string $w \in \Sigma^*$ and any states $p, q \in K$, $(q, w) \vdash^*_M (p, \varepsilon)$ iff $(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p. In other words, if the original machine M starts in state q and, after reading the string w, can land in state p, then the new machine M' must behave as follows: when started in the state that corresponds to the set of states the original machine M could get to from q without consuming any input, M' reads the string w and lands in one of its new "set" states that contains p. Furthermore, because of the only-if part of the lemma, M' must end up in a "set" state that contains only states that M could get to from q after reading w and following any available epsilon transitions.

If we assume that this lemma is true, then the proof that M' is equivalent to M is straightforward: Consider any string $w \in \Sigma^*$. If $w \in L(M)$ (i.e., the original machine M accepts w) then the following two statements must be true:

1. The original machine M, when started in its start state, can consume w and end up in a final state. This must be true given the definition of what it means for a machine to accept a string.

2. $(E(s), w) \vdash^*_{M'} (Q, \varepsilon)$ for some Q containing some $f \in F$. In other words, the new machine, when started in its start state, can consume w and end up in one of its final states. This follows from the lemma, which is more general and describes a computation from any state to any other. But if we use the lemma and let q equal s (i.e., M begins in its start state) and p = f for some $f \in F$ (i.e., M ends in a final state), then we have that the new machine M', when started in its start state, E(s), will consume w and end in a state that contains f. But if M' does that, then it has ended up in one of its final states (by the definition of K' in step 5 of the algorithm). So M' accepts w (by the definition of what it means for a machine to accept a string). Thus M' accepts precisely the same set of strings that M does.

Now all we have to do is to prove the lemma. What the lemma is saying is that we've built M' from M in such a way that the computations of M' mirror those of M and guarantee that the two machines accept the same strings. But of course we didn't build M' to perform an entire computation. All we did was to describe how to construct δ'. In other words, we defined how individual steps of the computation work. What we need to do now is to show that the individual steps, when taken together, do the right thing for strings of any length. The obvious way to do that, since we know what happens one step at a time, is to prove the lemma by induction on |w|.

We must first prove that the lemma is true for the base case, where |w| = 0 (i.e., w = ε). To do this, we actually have to do two proofs, one to establish the *if* part of the lemma, and the other to establish the *only if* part:

Basis step, *if* part: Prove $(q, w) \vdash^*_M (p, \varepsilon)$ if $(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p. Or, turning it around to make it a little clearer,

$[(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p $] \Rightarrow (q, w) \vdash^*_M (p, \varepsilon)$

If |w| = 0, then M' makes no moves. So it must end in the same state it started in, namely E(q). If we're told that it ends in some state that contains p, then $p \in E(q)$. But, given our definition of E(x), that means exactly that, in the original machine M, p is reachable from q just be following ε transitions, which is exactly what we need to show.

Basis step, *only if* part: Recall that *only if* is equivalent to implies. So now we need to show:

$[(q, w) \vdash^*_M (p, \varepsilon)] \Rightarrow (E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

If |w| = 0, and the original machine M goes from q to p with only w as input, it must go from q to p following just ε transitions. In other words $p \in E(q)$. Now consider the new machine M'. It starts in E(q), the set state that includes all the states that are reachable from q via ε transitions. Since the new machine is deterministic, it will make no moves at all if its input is ε. So it will halt in exactly the same state it started in, namely E(q). Since we know that $p \in E(q)$, we know that M' has halted in a set state that includes p, which is exactly what we needed to show.

Next we'll prove that if the lemma is true for all strings w of length k, $k \geq 0$, then it is true for all strings of length k + 1. Considering strings of length k + 1, we know that we are dealing with strings of a least one character. So we can rewrite any such string as zx, where x is a single character and z is a string of length k. The way that M and M' process z will thus be covered by the induction hypothesis. We'll use our definition of δ', which specifies how each individual step of M' operates, to show that, assuming that the machines behave correctly for the first k characters, they behave correctly for the last character also and thus for the entire string of length k + 1. Recall our definition of δ':

$\delta'(Q, a) = \cup\{E(p) : p \in K$ and $(q, a, p) \in \Delta$ for some $q \in Q\}$.

To prove the lemma, we must show a relationship between the behavior of:

M: $(q, w) \vdash^*_M (p, \varepsilon)$, and

M': $(E(q), w) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

Rewriting w as zx, we have

M: $(q, zx) \vdash^*_M (p, \varepsilon)$

M': $(E(q), zx) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

Breaking each of these computations into two pieces, the processing of z followed by the processing of x, we have:

M: $(q, zx) \vdash^*_M (s_i, x) \vdash^*_M (p, \varepsilon)$

M': $(E(q), zx) \vdash^*_{M'} (S, x) \vdash^*_{M'} (P, \varepsilon)$ for some $P \in K'$ that contains p

In other words, after processing z, M will be in some set of states s_i, and M' will be in some state, which we'll call S. Again, we'll split the proof into two parts:

Induction step, *if* part:

$$[\ (E(q), zx) \vdash^*_{M'} (S, x) \vdash^*_{M'} (P, \varepsilon) \text{ for some } P \in K' \text{ that contains } p\ ] \Rightarrow (q, zx) \vdash^*_{M} (s_i, x) \vdash^*_{M} (p, \varepsilon)$$

If, after reading z, M' is in state S, we know, from the induction hypothesis, that the original machine M, after reading z, must be in some set of states s_i and that S is precisely that set. Now we just have to describe what happens at the last step when the two machines read x. If we have that M', starting in S and reading x lands in P, then we know, from the definition of δ' above, that P contains precisely the states that M could land in after starting in any s_i and reading x. Thus if $p \in P$, p must be a state that M could land in.

Induction step, *only if* part:

$$(q, zx) \vdash^*_{M} (s_i, x) \vdash^*_{M} (p, \varepsilon) \Rightarrow (E(q), zx) \vdash^*_{M'} (S, x) \vdash^*_{M'} (P, \varepsilon) \text{ for some } P \in K' \text{ that contains } p$$

By the induction hypothesis, we know that if M, after processing z, can reach some set of states s_i, then S (the state M' is in after processing z) must contain precisely all the s_i's. Knowing that, and our definition of δ', we know that from S, reading x, M' must be in some set state P that contains precisely the states that M can reach starting in any of the s_i's, reading x, and then following all ε transitions. So, after consuming w (zx), M', when started in E(q) must end up in a state P that contains all and only the states p that M, when started in q, could end up in.

This theorem is a very useful result when we're dealing with FSAs. It's true that, by and large, we want to build deterministic machines. But, because we have provided a constructive proof of this theorem, we know that we can design a deterministic FSA by first describing a nondeterministic one (which is sometimes a much simpler task), and then applying our algorithm to construct a corresponding deterministic one.

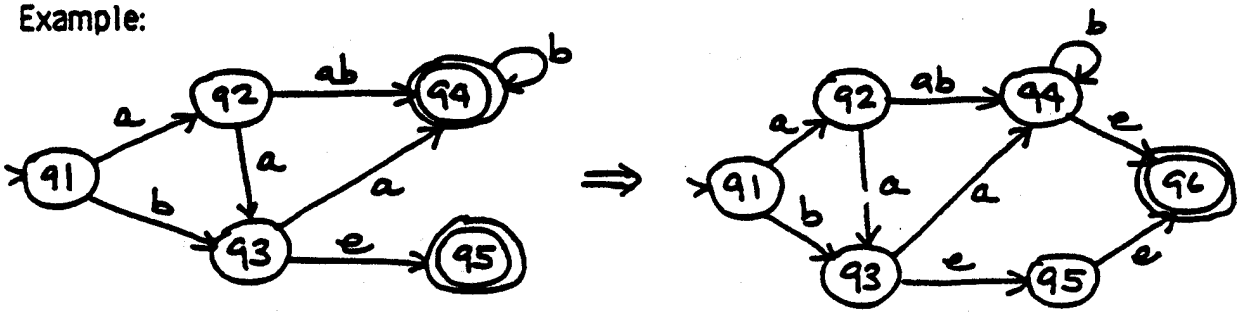
3 Generating Regular Expressions from Finite State Machines

1. Preparations (Note: FA may be non-deterministic in general)

If a) there is more than one final state
or b) there is just one final state but it lies on a loop,

then a) create a new final state
b) run e-transitions from the old final state(s) to the new one
and c) make the old final state(s) non-final

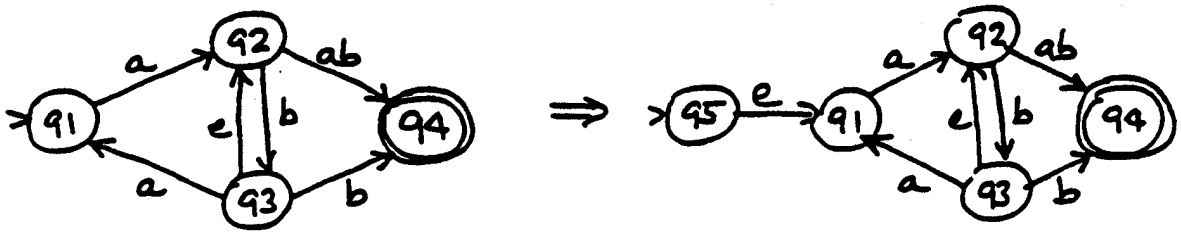
Example:



If the initial state is part of a loop,

then a) create a new initial state
b) run an e-transition from the new initial state to the old one
and c) make the old initial state non-initial

Example:



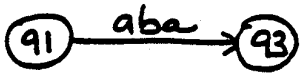
(Note: nothing needs to be done to the final state here because it does not lie on a loop. Similarly, nothing needs to be done to the initial state in the first example.)

II. Eliminating states

One by one, remove states which are neither initial nor final, replacing the connections between remaining states in such a way that the transitions are preserved. In general, the reconstructed transitions may be labelled with regular expressions rather than just by strings.

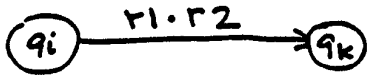
1. Example:

q2 can be eliminated and the connection between q1 and q3 becomes:

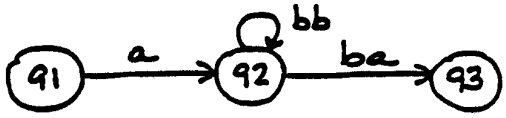


In general, if r1 and r2 are any regular expressions, produced perhaps by other steps in the algorithm, and occur in the configuration:

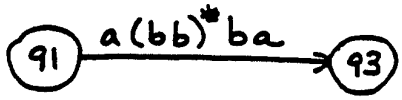
then this can be replaced by



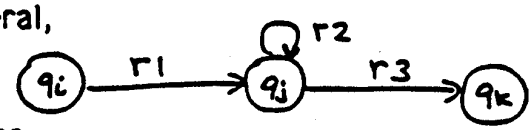
2) If the eliminated state happens to contain a "simple" loop, e.g.:



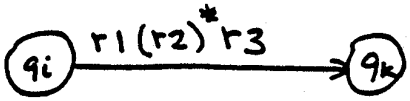
when q2 is eliminated, the transition becomes:



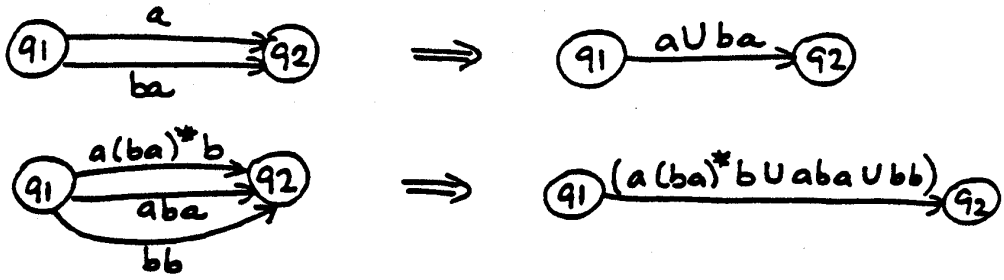
In general,



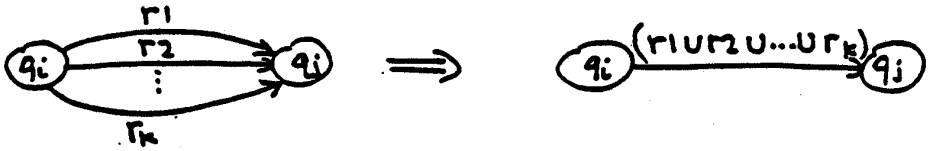
becomes



3) Parallel transitions can be coalesced into a single transition labelled by an expression which is the union of the originals:



In general:

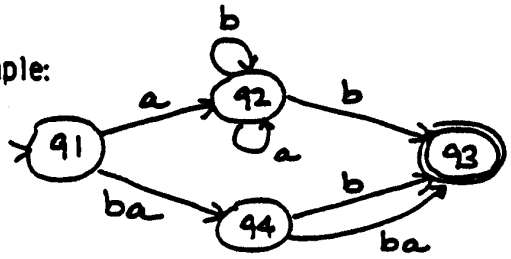


One proceeds in this manner, eliminating states one by one, until only a single transition remains connecting the initial and the one final state. The label on this transition is a regular expression for the original automaton.

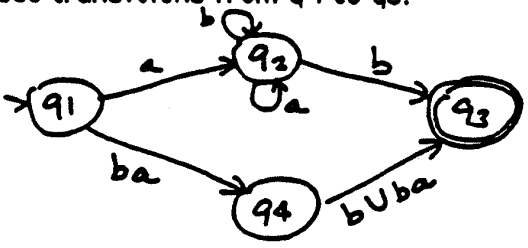
The order of elimination of states doesn't matter; equivalent regular expressions will be obtained so long as the procedure is done correctly.

Note that coalescing parallel transitions doesn't eliminate a state but reduces the number of transitions. In general, one should perform this step before eliminating a state that has parallel transitions into or out of it.

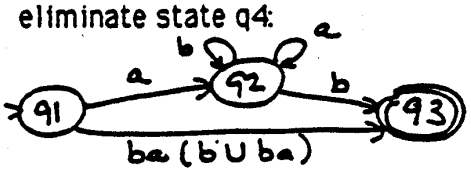
Example:



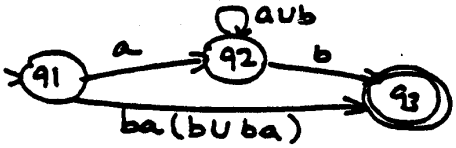
coalesce transitions from q4 to q3:



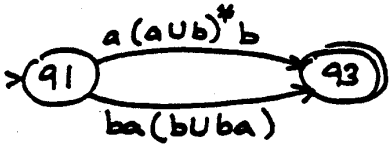
eliminate state q4:



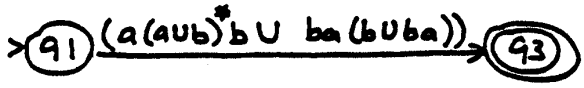
coalesce transitions from q2 to q2:



eliminate q2:

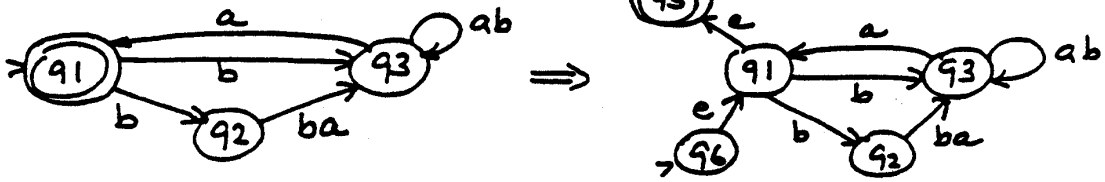


coalesce parallel transitions:

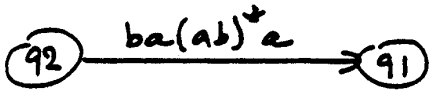


This is a regular expression for the FA. Check against the original.

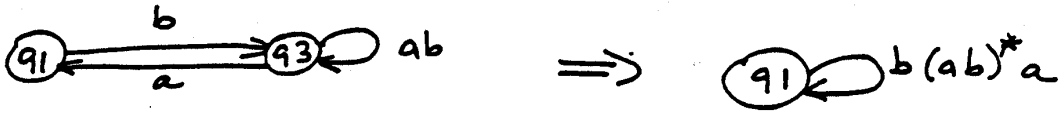
More complicated cases:



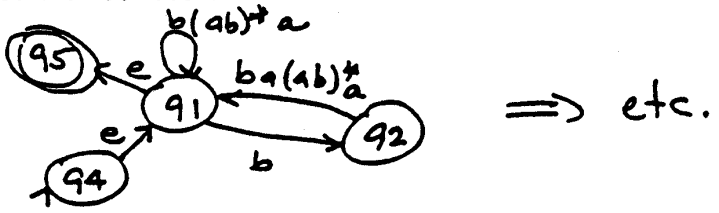
suppose we eliminate q3. Then the path from q2 to q1 becomes:



but we have also destroyed a path from q1 back to q1, namely:



so the net result is:



4 The Pumping Lemma for Regular Languages

4.1 What Does the Pumping Lemma Tell Us?

The pumping lemma is a powerful technique for showing that a language is **not** regular. The lemma describes a property that must be true of any language if it is regular. Thus, if we can show that some language L does not possess this property, then we know that L is not regular.

The key idea behind the pumping lemma derives from the fact that every regular language is recognizable by some FSM M (actually an infinite number of finite state machines, but we can just pick any one of them for our purposes here). So we know that, if a language L is regular, then every string s in L must drive M from the start state to some final state. There are only two ways this can happen:

1. s could be short and thus drive M from the start state to a final state without traversing any loops. By short we mean that if M has N states, then $|s| < N$. If s were any longer, M would have to visit some state more than once while processing s; in other words it would loop.
2. s could be long and M could traverse at least one loop. If it does that, then observe that another way to take M from the start state to the same final state would be to skip the loop. Some shorter string w in L would do exactly that. Still further ways to take M from the start state to the final state would be to traverse the loop two times, or three times, or any number of times. An infinite set of longer strings in L would do this.

Given some regular language L, we know that there exists an infinite number of FSMs that accept L, and the pumping lemma relies on the existence of at least one such machine M. In fact, it needs to know the number of states in M. But what if we don't have M handy? No problem. All we need is to know that M exists and that it has some number of states, which we'll call N. As long as we make no assumptions about what N is, other than that it is at least 1, we can forge ahead without figuring out what M is or what N is.

The pumping lemma tells us that if a language L is regular, then any sufficiently long (as defined above) string s in L must be "pumpable". In other words, there is some substring t of s that corresponds to a loop in the recognizing machine M. Any string that can be derived from s either by pumping t out once (to get a shorter string that will go through the loop one fewer times) or by pumping t in any number of times (to get longer strings that will go through the loop additional times) must also be in L since they will drive M from its start state to its final state. If we can show that there is even one sufficiently long string s that isn't pumpable, then we know that L must not be regular.

You may be wondering why we can only guarantee that we can pump out once, yet we must be able to pump in an arbitrary number of times. Clearly we must be able to pump in an arbitrary number of times. If there's a loop in our machine M, there is no limit to the number of times we can traverse it and still get to a final state. But why only once for pumping out? Sometimes it may in fact be possible to pump out more. But the lemma doesn't require that we be able to. Why? When we pick a string s that is "sufficiently long", all we know is that it is long enough that M must visit at least one state more than once. In other words, it must traverse at least one loop of length one at least once. It may do more, but we can't be sure of it. So the only thing we're guaranteed is that we can pump out the one pass through the loop that we're sure must exist.

Pumping Lemma:

If L is regular, then
  $\exists$ $N \geq 1$, such that
    $\forall$ strings w, where $|w| \geq N$,
      $\exists$ x, y, z, such that $w = xyz$, and
        $|xy| \leq N$, and
        $y \neq \varepsilon$, and
        $\forall$ $q \geq 0$, xy^qz is in L.

The lemma we've just stated is sometimes referred to as the Strong Pumping Lemma. That's because there is a weaker version that is much less easy to use, yet no easier to prove. We won't say anything more about it, but at least now you know what it means if someone refers to the Strong Pumping Lemma.

4.2 Using the Pumping Lemma

The key to using the pumping lemma correctly to prove that a language L is not regular is to understand the nested quantifiers in the lemma. Remember, our goal is to show that our language L fails to satisfy the requirements of the lemma (and thus is not regular). In other words, we're looking for a counterexample. But when do we get to pick any example we want and when do we have to show that there is no example? The lemma contains both universal and existential quantifiers, so we'd expect some of each. But the key is that we want to show that the lemma does not apply to our language L. So we're essentially taking the not of it. What happens when we do that? Remember two key results from logic:

$\neg\forall x\ P(x) = \exists x\ \neg P(x)$

$\neg\exists x\ P(x) = \forall x\ \neg P(x)$

So if the lemma says something must be true for all strings, we show that there exists at least one string for which it's false. If the lemma says that there exists some object with some properties, we show that all possible objects fail to have those properties. More specifically:

At the top level, the pumping lemma states

L regular $\Rightarrow$ $\exists N \geq 1$, P(L, N), where P is the rest of the lemma (think of P as the Pumpable property).

To show that the lemma does not correctly describe our language L, we must show

$\neg(\exists N\ N \geq 1, P(L, N))$, or, equivalently,

$\forall N\ \neg P(L, N)$

The lemma asserts first that there exists some magic number N, which defines what me mean by "sufficiently long." The lemma doesn't tell us what N is (although we know it derives from the number of states in some machine M that accepts L). We need to show that no such N exists. So we don't get to pick a specific N to work with. Instead:

We must carry N through the rest of what we do as a variable and make no assumptions about it.

Next, we must look inside the pumpable property. The lemma states that every string that is longer than N must be pumpable. To prove that that isn't true of L, all we have to do is to find a single long string in L that isn't pumpable. So:

We get to pick a string w.

Next, the lemma asserts that there is a way to carve up our string into substrings x, y, and z such that pumping works. So we must show that there is no such x, y, z triple. This is the tricky part. To show this, we must enumerate all logically possible ways of carving up w into x, y, and z. For each such possibility, we must show that at least one of the pumping requirements is not satisfied. So:

We don't get to pick x, y, and z. We must show that pumping fails for all possible x, y, z triples.

Sometimes, we can show this easily without actually enumerating ways of carving up w into x, y, and z. But in other cases, it may be necessary to enumerate two or more possibilities.

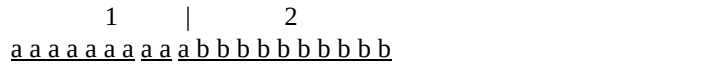
Let's look at an example. The classic one is L = $a^x b^x$ is not regular. Intuitively, we knew it couldn't be. To decide whether a string is in L, we have to count the a's, and then compare that number to the number of b's. Clearly no machine with a finite number of states can do that. Now we're actually in a position to prove this. We show that for any value of N we'll get a contradiction. We get to choose any w we want as long as it's length is greater than or equal to N. Let's choose w to be be $a^N b^N$. Next, we must show that there is no x, y, z triple with the required properties:

$|xy| \leq N$,

$y \neq \varepsilon$,

$\forall\ q \geq 0$, $xy^q z$ is in L.

Suppose there were. Then it might look something like this (this is just for illustration):



x y z

Don't take this picture to be making any claim about what x, y, and z are. But what the picture does show is that w is composed of two regions:

1. The initial segment, which is all a's.
2. The final segment, which is all b's.

Typically, as we attempt to show that there is no x, y, z triple that satisfies all the conditions of the pumping lemma, what we'll do is to consider the ways that y can be spread within the regions of w. In this example, we observe immediately that since $|xy| \leq N$, y must be a^g for some $g \geq 1$. (In other words, y must lie completely within the first region.) Now there's just one case to consider. Clearly we'll add just a's as we pump, so there will be more a's than b's, so we'll generate strings that are not in L. Thus w is not pumpable, we've found a contradiction, and L is not regular.

The most common mistake people make in applying the pumping lemma is to show a particular x, y, z triple that isn't pumpable. Remember, you must show that all such triples fail to be pumpable.

Suppose you try to apply the pumping lemma to a language L and you fail to find a counterexample. In other words, every string w that you examine is pumpable. What does that mean? Does it mean that L is regular? No. It's true that if L is regular, you will be unable to find a counterexample. But if L isn't regular, you may fail if you just don't find the right w. In other words, even in a non regular language, there may be plenty of strings that are pumpable. For example, consider $L = a^x b^x \cup a^y$. In other words, if there are any b's there must be the same number of them as there are a's, but it's also okay just to have a's. We can prove that this language is not pumpable by choosing $w = a^N b^N$, just as we did for our previous example, and the proof will work just as it did above. But suppose that were less clever. Let's choose $w = a^N$. Now again we know that y must be a^g for some $g \geq 1$. But now, if we pump y either in or out, we still get strings in L, since all strings that just contain a's are in L. We haven't proved anything.

Remember that, when you go to apply the pumping lemma, the one thing that is in your control is the choice of w. As you get experience with this, you'll notice a few useful heuristics that will help you find a w that is easy to work with:

1. Choose w so that there are distinct regions, each of which contains only one element of Σ. When we considered $L = a^x b^x$, we had no choice about this, since every element of L (except ε) must have a region of a's followed by a region of b's. But suppose we were interested in $L' = \{w \in \{a, b\}^*: w \text{ contains an equal number of a's and b's}\}$. We might consider choosing $w = (ab)^N$. But now there are not clear cut regions. We won't be able to use pumping successfully because if y = ab, then we can pump to our hearts delight and we'll keep getting strings in L. What we need to do is to choose $a^N b^N$, just as we did when we were working on L. Sure, L' doesn't require that all the a's come first. But strings in which all the a's do come first are fine elements of L', and they produce clear cut regions that make the pumping lemma useful.
2. Choose w so that the regions are big enough that there is a minimal number of configurations for y across the regions. In particular, you must pick w so that it has length at least N. But there's no reason to be parsimonious. For example, when we were working on $a^x b^x$, we could have chosen $w = a^{N/2} b^{N/2}$. That would have been long enough. But then we couldn't have known that y would be a string of a's. We would have had to consider several different possibilities for y. (You might want to work this one out to see what happens.) It will generally help to choose w so that each region is of length at least N.
3. Whenever possible, choose w so that there are at least two regions with a clear boundary between them. In particular, you want to choose w so that there are at least two regions that must be related in some way (e.g., the a region must be the same length as the b region). If you follow this rule and rule 2 at the same time, then you'll be assured that as you pump y, you'll change one of the regions without changing the other, thus producing strings that aren't in your language.

The pumping lemma is a very powerful tool for showing that a language isn't regular. But it does take practice to use it right. As you're doing the homework problems for this section, you may find it helpful to use the worksheet that appears on the next page.

Using the Pumping Lemma for Regular Languages

If L is regular, then

There exists an $N \geq 1$, (Just **call it N**) such that

for all strings w, where $|w| \geq N$,

(Since true for all w, it must be true for any particular one, so you **pick w**)

(Hint: describe w in terms of N)

there exist x, y, z, such that $w = xyz$

and $|xy| \leq N$,

and $y \neq \varepsilon$,

and for all $q \geq 0$, xy^qz is in L.

(Since must hold for all y, we **show that it can't hold for any y that meets the requirements: $|xy| \leq N$, and $y \neq \varepsilon$. To do this:**

Write out w:

List all the possibilities for y:

[1]

[2]

[3]

[4]

For each possibility for y, xy^qz must be in L, for all q. So:

For each possibility for y, find some value of q such that xy^qz is not in L. Generally q will be either 0 or 2.

	y	**q**
[1]		
[2]		
[3]		
[4]		

Q.E.D.