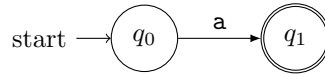


For all  $a \in \Sigma$ , the singleton language is recognized by the following deterministic finite automaton:



( $\Leftarrow$ ): This direction of the statement is trivially true, by Definition 10. □

### 3 Regular Expressions

If you frequently use a Unix-based system with a terminal, you may be familiar with utilities such as `grep`, which searches an input text file for lines that match a specified format. For example, on my computer, I can search the dictionary file (`/usr/share/dict/words`) for all words that contain “theory”:

```
taylor@SmithBook:~> grep theory /usr/share/dict/words
countertheory
theory
theoryless
theorymonger
```

But, to be fair, doing something like that is a bit overkill when one could just open the file in a text editor and use the Find tool to search for the word “theory”. Where `grep` really shines is when we need to search for text matching a *pattern*, like so:

```
taylor@SmithBook:~> grep ^u.*ity$ /usr/share/dict/words
ubiquity
ultimity
ultrafilterability
...
usability
utility
utterability
uxorality
```

In this example, I searched for all words in `/usr/share/dict/words` that began with a `u` and ended with `ity`, such as `university`. The ubiquity of this pattern in the English language is evident:

```
taylor@SmithBook:~> grep ^u.*ity$ /usr/share/dict/words | wc -l
235
```

Utilities like `grep` use patterns to perform fast searches in text files, and we can formalize the notion of a pattern in terms of our familiar regular operations. This formalization is known as a *regular expression*.

#### 3.1 Definition

To define regular expressions, let us recall our first characterization of regular languages from Definition 3. We stated that a language was regular if we could represent it in terms of five “components”: the empty language,  $\emptyset$ ; the singleton language  $\{a\}$  for all  $a \in \Sigma$ ; the union operation; the concatenation operation; and the Kleene star operation. To define regular expressions, we really don’t need to make any big changes to this characterization, apart from thinking about symbols instead of languages!

**Definition 25** (Regular expression). Let  $\Sigma$  be an alphabet. The class of regular expressions is defined inductively as follows:

1.  $r = \emptyset$  is a regular expression;
2.  $r = \epsilon$  is a regular expression;
3. For each  $a \in \Sigma$ ,  $r = a$  is a regular expression;
4. For regular expressions  $r_1$  and  $r_2$ ,  $r = r_1 + r_2$  is a regular expression;
5. For regular expressions  $r_1$  and  $r_2$ ,  $r = r_1 r_2$  is a regular expression; and
6. For a regular expression  $r$ ,  $r^*$  is a regular expression.

Every regular expression *represents* a language, and we denote the language represented by a regular expression  $r$  by  $L(r)$ . Note that each of the six “base” regular expressions correspond to their own language. If  $r = \emptyset$ , then  $L(r) = \emptyset$ . Likewise, if  $r = \epsilon$ , then  $L(r) = \{\epsilon\}$ , and if  $r = a$ , then  $L(r) = \{a\}$ . The remaining three regular expressions correspond exactly to the regular operations of union, concatenation, and Kleene star.

**Example 26.** Let  $\Sigma = \{a, b\}$ , and consider the language  $L_{\text{odda}} = \{w \mid w \text{ contains an odd number of } as\}$ . This language is represented by the regular expression  $r_{\text{odda}} = b^*(ab^*ab^*)^*ab^*$ . The first component of  $r$ ,  $b^*$ , recognizes zero or more leading bs. The middle component,  $(ab^*ab^*)^*$ , recognizes zero or more pairs of as, where each a is followed by zero or more bs. The last component,  $ab^*$ , recognizes an additional a to ensure the total number of as is odd, followed by zero or more bs.

Note that this regular expression is not unique; the same language is recognized by the regular expression  $r'_{\text{odda}} = b^*ab^*(ab^*ab^*)^*$ .

**Example 27.** Consider the regular expression  $r = (a \cup b)^*b$  over the alphabet  $\Sigma = \{a, b\}$ . We can “decompose” the language represented by  $r$  in the following way:

$$\begin{aligned} L(r) &= L((a \cup b)^*b) \\ &= L(a \cup b)^*L(b) \\ &= (L(a) \cup L(b))^*L(b) \\ &= (\{a\} \cup \{b\})^*\{b\} \\ &= \{a, b\}^*\{b\}. \end{aligned}$$

Therefore,  $r$  represents the language  $L = \{w \mid w \text{ ends with } b\}$ .

**Example 28.** The empty word  $\epsilon$  and the empty language  $\emptyset$  operate a little differently than other words and languages in regular expressions.

- For the empty word  $\epsilon$  and any regular expression  $r$ , we have that  $r\epsilon = r$  but  $r \cup \epsilon \neq r$  in general. We also have that  $\epsilon^* = \{\epsilon\}$ .
- For the empty language  $\emptyset$  and any regular expression  $r$ , we have that  $r \cup \emptyset = r$  but  $r\emptyset = \emptyset$ . We also have that  $\emptyset^* = \{\epsilon\}$ .

Just like how mathematics has an order of operations, regular expressions abide by their own order of precedence. The Kleene star is applied first, followed by concatenation, and then union. We can modify the order in which operations are applied by adding parentheses to a regular expression, which do not affect the language represented by that regular expression.

**Example 29.** Consider the regular expressions  $r_1 = 0 \cup 1^*10 \cup 1^*$  and  $r_2 = (0 \cup 1)^*1(0 \cup 1)^*$ . Clearly, the only visual difference between  $r_1$  and  $r_2$  is the addition of parentheses. However, the languages represented by  $r_1$  and  $r_2$  are quite different:

- The expression  $r_1$  represents the language containing the word 0, all words consisting of at least one 1 with one 0 at the end, and all words consisting of zero or more 1s.
- The expression  $r_2$  represents the language consisting of all words that contain at least one 1.

We may additionally define some shorthand notation to make our regular expressions look nicer, though strictly speaking, this notation is not “official”. We may choose to write  $r^+ = rr^* = r^*r$ , which is sometimes called the *Kleene plus* operation. Similarly, we may use exponents to denote iterated concatenation; that is,  $r^k$  denotes  $r$  concatenated with itself  $k$  times.

You may now be wondering why our earlier command line examples used pattern-matching symbols like  $\wedge$ ,  $.$ , and  $\$$ , when Definition 25 didn’t define any of those symbols. This is because our definition of a regular expression is the purely-theoretic definition, and as the name suggests, it is meant to coincide exactly with our definition of a regular language. It is therefore different from the practical implementation of “regular expressions”, where we can use special symbols to indicate the start or end of a word, match any symbol instead of one specific symbol, and make back-references among other things. The literature refers to these practical implementations as *extended regular expressions*, and these expressions can, in fact, represent more than just the class of regular languages. Thus, in the context of this lecture, when we refer to a regular expression we will be following Definition 25.

### 3.2 RE = DFA

At this point, we now have three ways to represent a regular language: by using regular operations, by constructing a finite automaton, and by writing a regular expression. In some cases, it’s easier for us to represent a regular language using a regular expression, while in other cases it might be easier for us to construct a finite automaton to recognize the language. However, is it always the case that, if we can do one, we can also do the other?

We already know from the last section that the first two representations are equivalent in terms of their descriptive power, so it would be nice to show that our new method of representation is similarly equivalent. By doing so, we can translate between all three representations. This brings us to the next exciting result of the lecture.

**Theorem 30.** *A language  $A$  is regular if and only if there exists a regular expression  $r$  such that  $L(r) = A$ .*

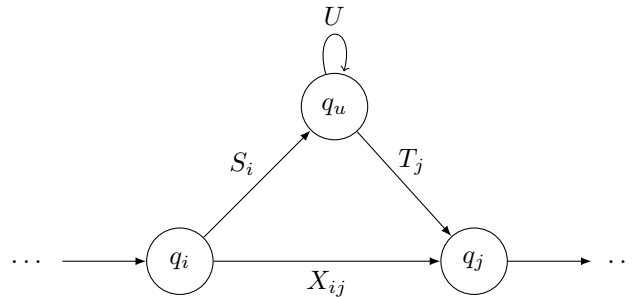
*Proof.* ( $\Rightarrow$ ): To prove this direction of the statement, we will take a deterministic finite automaton recognizing the language  $A$ , and then convert the finite automaton to a regular expression. We will use a *state elimination algorithm* to perform this conversion.

Note that, for this proof only, we will assume that the transitions of our finite automaton can be labelled by regular expressions and not just symbols.

Suppose that we are given a deterministic finite automaton  $\mathcal{M}$  such that  $L(\mathcal{M}) = A$ . Further suppose, without loss of generality, that there exists at most one transition between any two states of  $\mathcal{M}$ ; we can make this assumption since multiple transitions between two states on symbols  $a_1, \dots, a_n$  can be replaced by the single transition on the regular expression  $a_1 \cup \dots \cup a_n$ .

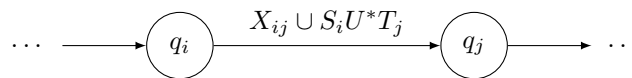
If  $\mathcal{M}$  contains no accepting state, then  $A = \emptyset$  and we are done. Otherwise, if  $\mathcal{M}$  contains multiple accepting states, convert them to non-accepting states and add epsilon transitions from each formerly-accepting state to a new single accepting state. If the initial state is also an accepting state, make a similar change to the initial state.

Now, we eliminate all states  $q_u$  of  $\mathcal{M}$  that are neither initial nor accepting. Suppose that  $\mathcal{M}$  contains the following substructure:



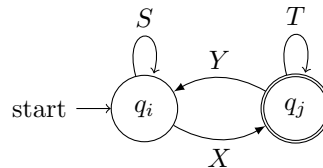
In this substructure, all transitions from states  $q_i \neq q_u$  to state  $q_u$  are labelled by a regular expression  $S_i$ ; all transitions from state  $q_u$  to states  $q_j \neq q_u$  are labelled by a regular expression  $T_j$ , and for all such states  $q_i$  and  $q_j$  the transition between these states is labelled by a regular expression  $X_{ij}$ , or  $\emptyset$  if no such transition exists. Lastly, any loop from  $q_u$  to itself is labelled by a regular expression  $U$ , or  $\emptyset$  if no loop exists.

We may eliminate state  $q_u$  from  $\mathcal{M}$  as follows: for each pair of states  $q_i$  and  $q_j$ , the regular expression  $X_{ij}$  on the transition is replaced by  $X_{ij} \cup S_i U^* T_j$ .



We then repeat this procedure for all non-initial and non-accepting states until the only states remaining in the finite automaton are the single initial and accepting states.

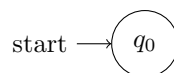
Suppose that, at this stage of the algorithm, our finite automaton is of the following form, where  $S$ ,  $T$ ,  $X$ , and  $Y$  are regular expressions:



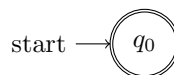
If any of these transitions do not exist, then we simply add them to the finite automaton labelled by  $\emptyset$ . Then the language recognized by  $\mathcal{M}$  is represented by the regular expression  $S^* X (T \cup Y S^* X)^*$ .

( $\Leftarrow$ ): To prove this direction of the statement, we will convert a regular expression  $r$  to a nondeterministic finite automaton  $\mathcal{M}$  using a construction known as the *McNaughton–Yamada–Thompson algorithm*. We consider each of the “base” regular expressions:

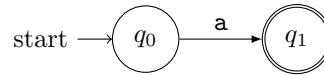
1. If  $r = \emptyset$ , then  $L(r) = \emptyset$  and this language is recognized by the following nondeterministic finite automaton:



2. If  $r = \epsilon$ , then  $L(r) = \{\epsilon\}$  and this language is recognized by the following nondeterministic finite automaton:



- If  $r = a$  for some  $a \in \Sigma$ , then  $L(r) = \{a\}$  and this language is recognized by the following nondeterministic finite automaton:

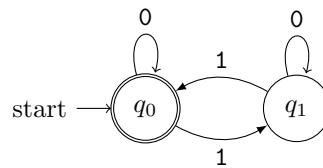


- If  $r = r_1 \cup r_2$  for some regular expressions  $r_1$  and  $r_2$ , then the corresponding language is recognized by the nondeterministic finite automaton constructed in the proof of Theorem 21.
- If  $r = r_1 r_2$  for some regular expressions  $r_1$  and  $r_2$ , then the corresponding language is recognized by the nondeterministic finite automaton constructed in the proof of Theorem 22.
- If  $r = r^*$  for some regular expression  $r$ , then the corresponding language is recognized by the nondeterministic finite automaton constructed in the proof of Theorem 23.

Since, in each case, we can convert the “base” regular expression to a nondeterministic finite automaton, we know by Theorem 24 that the language  $L(r)$  is regular.  $\square$

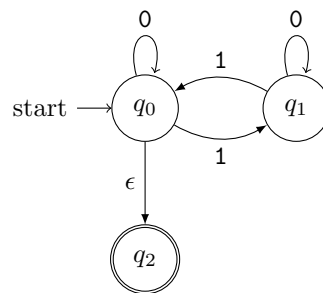
As an illustration of the state elimination algorithm we used in one direction of our proof, let us consider a small example of converting a deterministic finite automaton to a regular expression.

**Example 31.** Consider the following deterministic finite automaton  $\mathcal{M}$ :

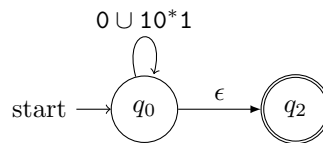


This finite automaton recognizes the language  $L(\mathcal{M}) = \{w \mid w \text{ contains an even number of 1s}\}$ .

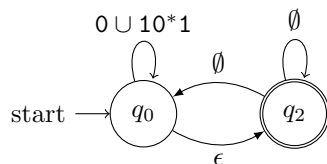
Since the initial state of  $\mathcal{M}$  is also an accepting state, we begin by creating a new accepting state, converting the initial state to be nonaccepting, and then adding an epsilon transition from the initial state to our new accepting state.



We now use our state elimination algorithm to remove  $q_1$ , which is the only state that is neither initial nor accepting. There exists a single transition from  $q_0$  to  $q_1$  and a single transition from  $q_1$  to  $q_0$ . Let  $S_0 = 1$ ,  $T_0 = 1$ ,  $X_{00} = 0$ , and  $U = 0$ . We can then eliminate the state  $q_1$  by relabelling the loop on  $q_0$  to use the regular expression  $X_{00} \cup S_0 U^* T_0 = 0 \cup 10^*1$ .



We add the missing transitions to obtain a finite automaton of the form specified in the proof of Theorem 30:



Consequently, the regular expression corresponding to this finite automaton is  $(0 \cup 10^*1)^* \epsilon (\emptyset \cup \emptyset (0 \cup 10^*1)^* \epsilon)^*$ , which simplifies to  $(0 \cup 10^*1)^*$ .

Putting together all of our equivalencies, we get the following important theorem.

**Theorem 32** (Kleene’s theorem). *A language  $R$  is regular if it satisfies any of the following equivalent properties:*

1. *There exists a deterministic finite automaton  $\mathcal{M}_D$  with  $L(\mathcal{M}_D) = R$ ;*
2. *There exists a nondeterministic finite automaton  $\mathcal{M}_N$  with  $L(\mathcal{M}_N) = R$ ; or*
3. *There exists a regular expression  $r$  with  $L(r) = R$ .*

## 4 Proving a Language is Nonregular

By now, it should be evident that finite automata and regular expressions are nice models to use when discussing computation in the abstract. They’re easy to define, easy to reason about, and they have a lot of nice properties that we can use in proofs. However, they are not the be-all and end-all of theoretical computer science. (Otherwise, this would be a rather short course!)

Both finite automata and regular expressions suffer the drawback of not having any way to store or recall data. Finite automata don’t have any storage mechanism, and regular expressions don’t allow for lookback. As we said in the section introducing finite automata, once the finite automaton reads a symbol and transitions to a state, it can never return to that symbol. For all intents and purposes, the symbol is lost forever, and the finite automaton doesn’t even remember having read it. Likewise, once a regular expression matches a symbol in a word and moves on to the next symbol, it has no way of remembering any previous symbols that were matched.

Naturally, this means that there exist some languages that cannot be recognized by a finite automaton (or, equivalently, represented by a regular expression), and therefore such languages cannot be regular. For instance, this is the canonical example of a language that no finite automaton can recognize:

$$L_{a=b} = \{a^n b^n \mid n \geq 0\}.$$

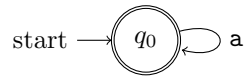
In this language, every word has an equal number of **a**s and **b**s, and all occurrences of **a** appear before the first occurrence of **b**. Some examples of words in this language are **ab**, **aaabbb**, **aaaaabbbbb**, and  $\epsilon$ .

Why can’t any finite automaton recognize this language? Because of that word *finite*. A finite automaton consists of a finite number of states, but in order to recognize this language, we would need to add a “chain” consisting of  $2n$  states to accept the word  $a^n b^n$  for every  $n \geq 0$ . Since  $n$  has no upper bound, we would need an infinite number of such “chains”, and therefore an infinite number of states! No finite automaton can recognize this language, because no finite automaton has a way of keeping track of the value  $n$  or counting the symbols using only a finite number of states.

However, we can’t totally rely on the claim that a finite automaton is incapable of recognizing a language if it has to count symbols. For instance, consider the language

$$L_a = \{a^n \mid n \geq 0\}.$$

This language contains an infinite number of words: one word for each  $n \geq 0$ , exactly like in  $L_{a=b}$ . But it's easy for a finite automaton to recognize  $L_a$ , and using only one state!



Thus, it should hopefully be clear that we need to take a slightly more intricate approach in order to prove a language is not regular. There are many more nonregular languages than there are regular languages, so instead of focusing on some sort of property that a nonregular language might have, let's instead find a property every regular language must have. We can then prove a language is nonregular by showing that the language *doesn't* have that property.

### 4.1 The Pumping Lemma for Regular Languages

The property of regular languages that we will make use of is the following: for every regular language, if we take a word in the language of sufficient length, then we can repeat (or *pump*) a middle portion of that word an arbitrary number of times and produce a new word that belongs to the same regular language. This fact, known as the *pumping lemma* for regular languages, allows us to prove a language is nonregular by contradiction; that is, by assuming the language is regular and pumping some sufficiently-long word to produce a word that does not belong to the language.

The formal statement of the pumping lemma is as follows.

**Lemma 33** (Pumping lemma for regular languages). *For all regular languages  $L$ , there exists  $p \geq 1$  where, for all  $w \in L$  with  $|w| \geq p$ , there exists a decomposition of  $w$  into three parts  $w = xyz$  such that*

1.  $|y| > 0$ ;
2.  $|xy| \leq p$ ; and
3. for all  $i \geq 0$ ,  $xy^iz \in L$ .

*Proof.* Let  $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$  be a deterministic finite automaton recognizing the language  $L$ , and let  $p$  denote the number of states of  $\mathcal{M}$ .

Take a word  $w = w_1w_2 \dots w_n$  of length  $n$  from  $L$ , where  $n \geq p$ , and let  $r_1, \dots, r_{n+1}$  be the accepting computation of  $\mathcal{M}$  on  $w$ . Specifically, let  $r_{i+1} = \delta(r_i, w_i)$  for all  $1 \leq i \leq n$ . Clearly, this accepting computation has length  $n + 1 \geq p + 1$ .

By the pigeonhole principle, there must exist at least two states in the first  $p + 1$  states of the accepting computation that are the same. Say that the first occurrence of the same state is  $r_j$  and the second occurrence is  $r_\ell$ . Since  $r_\ell$  occurs within the first  $p + 1$  states of the accepting computation, we know that  $\ell \leq p + 1$ .

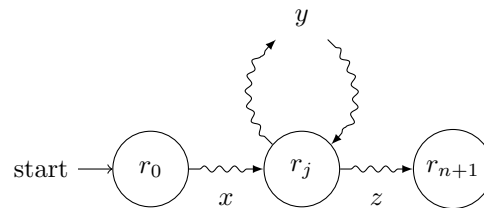
Decompose the word  $w$  into parts  $x = w_1 \dots w_{j-1}$ ,  $y = w_j \dots w_{\ell-1}$ , and  $z = w_\ell \dots w_n$ . As the part  $x$  is read,  $\mathcal{M}$  transitions from state  $r_1$  to state  $r_j$ . Likewise, as  $y$  is read,  $\mathcal{M}$  transitions from  $r_j$  to  $r_j$ , and as  $z$  is read,  $\mathcal{M}$  transitions from  $r_j$  to  $r_{n+1}$ . Since we are considering an accepting computation,  $r_{n+1}$  is a final state, and so  $\mathcal{M}$  must accept the word  $xy^iz$  for all  $i \geq 0$ . Moreover, we know that  $j \neq \ell$ , so  $|y| > 0$ . Lastly, since  $\ell \leq p + 1$ , we have that  $|xy| \leq p$ . Therefore, all three conditions of the pumping lemma are satisfied.  $\square$

Clearly, the pumping lemma contains a lot of notation and terminology to take in at once—not to mention four alternating quantifiers in a row! Let's break it down piece-by-piece to see what the lemma tells us.

- For all regular languages  $L$ ,  
 We can take any regular language  $L$ , and it will satisfy the pumping lemma.
- there exists  $p \geq 1$   
 Depending on the language  $L$  we consider, there exists a constant  $p$  for that language. We call  $p$  the *pumping constant*. (If you're curious,  $p$  is the number of states in the finite automaton recognizing  $L$ .)

- *where, for all  $w \in L$  with  $|w| \geq p$ ,*  
 We can take any word from  $L$  with length at least  $p$ , and it will satisfy the pumping lemma.
- *there exists a decomposition of  $w$  into three parts  $w = xyz$*   
 Depending on the word  $w$  we choose, we are able to decompose  $w$  into three parts:  $x$ ,  $y$ , and  $z$ . The  $y$  part is what we will use to do the pumping; the  $x$  and  $z$  parts are just the start and end parts of  $w$  that don't get pumped.
- *such that 1.  $|y| > 0$ ;*  
 This condition ensures that the  $y$  part of  $w$  is nonempty, so that we have something to pump.
- *2.  $|xy| \leq p$ ;*  
 This condition ensures that there exists some state in the finite automaton recognizing  $L$  that is visited more than once, and furthermore, we will visit that state during the computation before we finish reading the part  $y$ . (This condition is essentially an application of the pigeonhole principle.)
- *and 3. for all  $i \geq 0$ ,  $xy^i z \in L$ .*  
 This is the actual pumping part of the pumping lemma. This condition ensures that, no matter how many copies of the  $y$  part we include in our word (even zero copies), the resulting word will still belong to the language.

Diagrammatically, the proof of the pumping lemma can be reasoned about in the following way. All of the states of the finite automaton between  $r_0$  and  $r_j$  are used to read the part  $x$ , all of the states between  $r_j$  and  $r_{n+1}$  are used to read the part  $z$ , and there exists a loop of states that both starts and ends with  $r_j$  that is used to read the part  $y$ . We can take this loop as many times as we want while reading the input word, and taking one journey around the loop corresponds to “pumping” the word once.



Alternatively, we can think of the pumping lemma as an adversarial game, where we're trying to show that some language  $L$  is nonregular while our opponent is trying to show that  $L$  is, in fact, regular. If we win the game, then  $L$  is nonregular, while if our opponent wins, then  $L$  is regular. The rules of this game are as follows, so that you can play it at the next party you attend:

**Rules of the Pumping Lemma Game**

1. Your opponent chooses  $p \geq 1$ , and they claim it is the pumping constant for  $L$ .
2. You choose a word  $w \in L$  with  $|w| \geq p$ , and you claim this word can't be decomposed into parts  $w = xyz$  that satisfy the three conditions of the pumping lemma.
3. Your opponent chooses a decomposition  $w = xyz$  such that  $|y| \geq 0$  and  $|xy| \leq p$ , satisfying the first two conditions automatically, and they claim that this decomposition will also satisfy the third condition.
4. You choose  $i \geq 0$  such that  $xy^i z \notin L$ .

If you complete Step 4, then you win the game! Otherwise, if you can't find such an  $i$ , then you lose. Finally, if any of the claims in Steps 1-3 are false, then the person who made the claim loses.

Even though the pumping lemma looks complex, if we reduce it to a series of steps as we did here, then any proof showing that a language is nonregular simply has to follow each of the steps. As a result, nonregularity proofs tend to all have a similar structure. Let's take a look at an example of a pumping lemma proof using our canonical nonregular language,  $L_{a=b}$ .



**Example 34.** Let  $\Sigma = \{a, b\}$ , and consider the language  $L_{a=b} = \{a^n b^n \mid n \geq 0\}$ . We will use the pumping lemma to show that this language is nonregular.

Assume by way of contradiction that the language is regular, and let  $p$  denote the pumping constant given by the pumping lemma. We choose the word  $w = a^p b^p$ . Clearly,  $w \in L_{a=b}$  and  $|w| \geq p$ . Thus, there exists a decomposition  $w = xyz$  satisfying the three conditions of the pumping lemma.

We consider three cases, depending on the contents of the part  $y$  of the word  $w$ :

1. The part  $y$  contains only **as**. In this case, pumping  $y$  once to obtain the word  $xy^2z$  results in the word containing more **as** than **bs**, and so  $xy^2z \notin L_{a=b}$ . This violates the third condition of the pumping lemma.
2. The part  $y$  contains only **bs**. In this case, since the first  $p$  symbols of  $w$  are **as**, we must have that  $|xy| > p$ . This violates the second condition of the pumping lemma.
3. The part  $y$  contains both **as** and **bs**. Again, in this case, since the first  $p$  symbols of  $w$  are **as**, we must have that  $|xy| > p$ . This violates the second condition of the pumping lemma.

In all cases, one of the conditions of the pumping lemma is violated. As a consequence, the language cannot be regular.

A language doesn't necessarily have to count symbols in order to be nonregular. Since finite automata don't have any form of storage, they can't remember symbols they read earlier in an input word. This means that finite automata can't recall parts of a word, and so they can't recognize languages like  $L_{\text{double}} = \{ww \mid w \in \Sigma^*\}$ . Here, we prove that a similar language is nonregular: the language of palindromes,  $ww^R$ . (The notation  $w^R$  denotes the *reversal* of the word  $w$ .) Palindromes are words that read the same backward as they do forward.

**Example 35.** Let  $\Sigma = \{a, b\}$ , and consider the language  $L_{\text{pal}} = \{ww^R \mid w \in \Sigma^*\}$ . We will use the pumping lemma to show that this language is nonregular.

Assume by way of contradiction that the language is regular, and let  $p$  denote the pumping constant given by the pumping lemma. We choose the word  $w = a^p b b a^p$ . Clearly,  $w \in L_{\text{pal}}$  and  $|w| \geq p$ . Thus, there exists a decomposition  $w = xyz$  satisfying the three conditions of the pumping lemma.

Since the second condition of the pumping lemma tells us that  $|xy| \leq p$ , it must be the case that, in any decomposition, we have  $xy = a^k$  for some  $k \leq p$ . Consequently, we have  $y = a^\ell$  for some  $1 \leq \ell \leq k$ .

If we pump  $y$  once to obtain the word  $xy^2z$ , then we obtain the word  $a^{p+\ell} b b a^p$ , which is no longer a palindrome. This violates the third condition of the pumping lemma. As a consequence, the language cannot be regular.

Lastly, recall the third condition of the pumping lemma: for all  $i \geq 0$ ,  $xy^i z \in L$ . The third condition allows us not only to pump *up* by adding copies of  $y$  to the word, but also to pump *down* by removing  $y$  from the word. In some cases, pumping down can help us to prove a language is nonregular.

**Example 36.** Let  $\Sigma = \{a, b\}$ , and consider the language  $L_{a>b} = \{a^i b^j \mid i > j\}$ . We will use the pumping lemma to show that this language is nonregular.

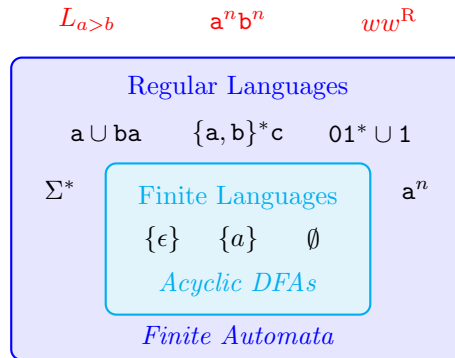
Assume by way of contradiction that the language is regular, and let  $p$  denote the pumping constant given by the pumping lemma. We choose the word  $w = a^{p+1} b^p$ . Clearly,  $w \in L_{a>b}$  and  $|w| \geq p$ . Thus, there exists a decomposition  $w = xyz$  satisfying the three conditions of the pumping lemma.

Since the second condition of the pumping lemma tells us that  $|xy| \leq p$ , it must be the case that, in any decomposition, we have  $xy = a^k$  for some  $k \leq p$ . Consequently, we have  $y = a^\ell$  for some  $1 \leq \ell \leq k$ .

If we pump  $y$  one or more times, then we will always end up with a word that contains more **as** than **bs**, and this word will always belong to the language  $L_{a>b}$ .

However, if we pump  $y$  down to obtain the word  $xy^0z = xz$ , then our word will be of the form  $a^{p+1-\ell}b^p$ . Since  $\ell \geq 1$ , our resultant word has at most as many  $a$ s as  $b$ s, and so it no longer belongs to the language  $L_{a>b}$ . This violates the third condition of the pumping lemma. As a consequence, the language cannot be regular.

Now that we've established that there exist both regular languages and nonregular languages, we can draw a diagram to represent the theory world as we know it so far. For the time being, we're only familiar with two language classes: the class of regular languages and the class of finite languages, which is a subclass of the regular languages that we mentioned very briefly. We also only know about one machine model: finite automata.<sup>3</sup> As a result, our diagram admittedly isn't very interesting right now, but as we continue in the course, we will expand and add to it.



<sup>3</sup>Finite languages, being a subclass of the regular languages, are recognized by a special kind of deterministic finite automaton with no cycles.