# ECS 120 Lesson 9 – Non-Regular Languages, the Pumping Lemma

Oliver Kreylos

Wednesday, April 18th, 2001

Having spent so much time on regular languages, on different ways to specify them, on operations on them and their closure under these operations, one could get the impression that all languages we are concerned about are regular. Far from the truth – though regular languages are important in a plethora of applications, the *really* interesting languages are not regular. Here are some examples of non-regular languages:

- $\left\{ w \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; w \text{ contains an equal number of } \mathtt{0}\text{s and } \mathtt{1}\text{s} \right\}$
- $\left\{ \mathtt{0}^n\mathtt{1}^m \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; n = m \geq 0 \right\}$
- $\left\{ \mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^* \;\middle|\; i + j = k \right\}$
- $\left\{ \mathtt{a}^p \in \{\mathtt{a}\}^* \;\middle|\; p \geq 2 \text{ is a prime number} \right\}$
- $\left\{ w \in \Sigma^* \;\middle|\; w \text{ is a palindrome} \right\}$
- $\left\{ w \in \Sigma^* \;\middle|\; w = xx \text{ for some } x \in \Sigma^* \right\}$
- $\left\{ w \in \big(\Sigma \cup \{\emptyset, \epsilon, \cup, {}^*, (, )\}\big)^* \;\middle|\; w \in \mathcal{R}(\Sigma) \right\}$

From these examples, we can boldly conclude that finite state machines have trouble recognizing languages that involve counting, calculating, storing input strings, and languages that are defined in a recursive fashion, e. g., regular expressions or arithmetic expressions. This intuition sometimes leads in the wrong direction, however: The language $\left\{ w \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; w \text{ has an equal number of occurrences of the substrings } \mathtt{01} \text{ and } \mathtt{10} \right\}$ is in fact regular.

With intuition seemingly at a loss, how can we decide whether a given language is regular or not? We already know a way of proving that a language is regular: If we are able to construct a finite state machine or regular expression that accepts/generates it, we know the language must be regular. On the other hand, if we are not able to construct a machine, we cannot conclude that the language is not regular (we might just not have found the correct one). To prove the non-regularity of a language $A$, we need deeper insight into the workings of automata, to show that there can be no automaton that accepts $A$. This insight comes in the form of a general theorem about the languages accepted by finite automata.

# 1 The Pumping Lemma

The pumping lemma is a theorem about regular languages. In other words, if a language is regular, it must behave according to the pumping lemma. We can use this to disprove the regularity of a language: If we can show that it violates the pumping lemma, it cannot be regular.

To understand the pumping lemma, we have to have a closer look at how any finite automaton accepts a word from its language. Let $M = (Q, \Sigma, \delta, q_0, F)$ be any DFA, and let $A = L(M)$ be its language. Now let $w = w_1 w_2 \dots w_n \in A$ be any word of length $n \geq 0$. We know that there must be a sequence of states $(q_0, q_1, \dots, q_n) \in Q^{n+1}$, such that $q_0$ is the start state, $q_n$ is a final state, and the labels on the transitions from $q_{i-1}$ to $q_i$ spell out the word $w$: $\forall i \in \{1, \dots, n\} : \delta(q_{i-1}, w_i) = q_i$, see Figure 1.

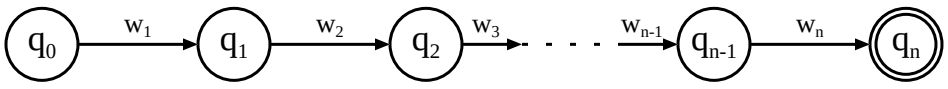


Figure 1: The chain of states $(q_0, q_1, \dots, q_n)$ that machine $M$ follows when accepting word $w = w_1 w_2 \dots w_n$.

Now, what happens if one state $q$ appears more than once in the chain of states, say $q = q_i$ and $q = q_j$, with $i < j$? In that case, the chain has a loop, and we can split the chain into three parts, see Figure 2:

1. The first part, called *prefix*, goes from state $q_0$ to $q_i$, the first occurrence of the double state $q$. This part spells out the word $x := w_1 \dots w_i$.

2. The second part, called *loop*, connects the two occurrences of $q$; it goes from $q_i$ to $q_j$, and spells out the word $y := w_{i+1} \dots w_j$. Since $i < j$, the length of $y$ must be at least one, $|y| \geq 1$.

3. The third part , called *suffix*, goes from state $q_j$ to the last state $q_n$ and spells out the word $z := w_{j+1} \dots w_n$.

By splitting the chain into three parts, the input word $w$ is split accordingly into three parts $x$, $y$ and $z$, such that $w = xyz$.

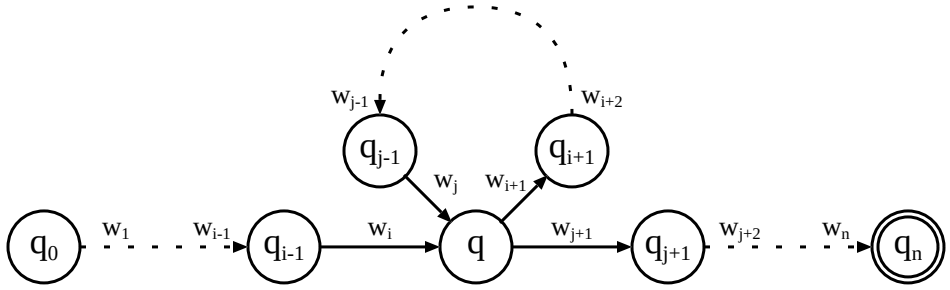


Figure 2: A state chain with a state $q$ occurring twice, at positions $i$ and $j$ in the chain. The chain forms a loop and can be split into three parts: The initial part (prefix), the loop part, and the final part (suffix).

From our definition of computation, we know that the only "memory" a finite automaton has about the current state of computation is the state it is currently in. The machine has no recollection of which states it visited earlier, or how it reached the current state. This means, whenever in the computation for the word $w$ the machine reaches state $q$, it does not now if it just arrived there at the end of the prefix, or if it already processed the loop part. Since the machine cannot distinguish these cases, any word $q' \in \Sigma^*$ that leads to state $q$, i. e., starts with a prefix $x$, then takes any number of turns around the loop back to state $q$, each time spelling out the word $y$, and then leads to state $q_n$, spelling out the suffix $z$, must be also be accepted by the machine. More formally, if $w = xyz \in A$, then for all $i \geq 0 : xy^iz \in A$.

By this reasoning, we know that if a regular language contains a word $w$ that has a loop in its computation chain,i. e., $w = xyz$, we can construct infinitely many other words which must be in the language as well, by "pumping" the loop part of the word any number of times, yielding $w' = xy^iz$. But how do we know if a word has a loop in its computation chain, if we do not know the exact structure of the machine that processes it?

All finite state machines have one thing in common: The set of states must be finite. If $A$ is a regular language, and $M = (Q, \Sigma, \delta, q_0, F)$ is a machine that accepts it, let us assume that $|Q| = p > 0$ is some constant. Now let us further assume that $A$ contains a word $w = w_1 \dots w_n \in A$ that has at least $p$ characters, $n \geq p$. This means the computation chain for $w$ must have $n$ transitions and $n + 1$ states. In other words, the computation chain contains more states than there are states in $Q$. From this follows that at least one state $q \in Q$ must occur twice in $w$'s computation chain. This rather obvious conclusion is called the *Pigeonhole Principle*: If there are more pigeons than pigeonholes, at least one pigeonhole must contain more than one pigeon. Interestingly enough, the Pigeonhole Principle is not true if there are infinitely many pigeonholes and pigeons, but that is a different story. The number of pigeonholes for a finite state machine, $|Q|$, is by definition always finite.

So now we know that every word $w$ that has at least as many characters as the machine that accepts it has states must have a loop in its computation chain, which means pumping the loop part will generate other words which must all be in the same language. The last question remaining is: When does the loop occur, or more accurately, what is the range of values for $i$ and $j$? By the Pigeonhole Principle, any chain of states containing $p+1$ states must already have a double state and hence a loop. In other words, $0 \leq i < j \leq p$. A chain of $p + 1$ states spells out $p$ characters; this means, the length of $x$ and $y$ combined can be at most $p$: $|xy| \leq p$.

We are now ready to combine all of the above observations, and to state the Pumping Lemma:

**Lemma 1 (Pumping Lemma)** *Let $A \subset \Sigma^*$ be a regular language over $\Sigma$. Then there exists a number $p > 0$, the* pumping length*, such that every word $w \in A$, with $|w| \geq p$, can be divided into three parts $w = xyz$, satisfying the following three conditions:*

1. $|y| \geq 1$,
2. $|xy| \leq p$, *and*
3. $\forall i \geq 0 : xy^i z \in A$.

# 2 Using the Pumping Lemma

The typical application for the Pumping Lemma is proving that a certain language is *not* regular. This is usually done in the following way:

1. Assume that the language is regular (to form a contradiction later).
2. If the language is regular, the Pumping Lemma applies to it. This means, there must be a pumping length $p$.
3. Pick any word $w$ from the language that is at least $p$ characters long.
4. For any possible way of splitting $w$ into parts $w = xyz$, show that at least one word $xy^iz$ is not in the language.
5. If this is the case, we have a contradiction with the assumption that the language was regular. Therefore it is not regular.

The tricky part about proofs like this is step 4. It is not enough to show that for *some* split of $w$ pumping is not possible – it has to be shown that there can be no split at all that allows pumping. Usually this is done by considering all possible cases that can arise from splitting a word, and showing that pumping does not work for each case separately.

## 2.1 The Language $\left\{ \mathtt{0}^n\mathtt{1}^m \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; n = m \geq 0 \right\}$

Let us use the Pumping Lemma to show that this language is not regular. First, we assume it is regular and that the Pumping Lemma applies, giving us a pumping length $p$. Now we take any word $w = \mathtt{0}^i\mathtt{1}^i$ that is longer than $p$. There are three different ways to split $w$:

1. $y = \mathtt{0}^n$, i. e., $y$ consists only of zeros. In that case, pumping is not possible: Since $|y| \geq 1$, any pumping will increase the number of zeros, but not the number of ones. Therefore, none of the pumped words can be in the language.
2. $y = \mathtt{1}^n$, i. e., $y$ consists only of ones. In that case, pumping is not possible, either: Since $|y| \geq 1$, any pumping will increase the number of ones, but not the number of zeros. Therefore, none of the pumped words can be in the language.

3. $y = \mathtt{0}^n\mathtt{1}^m$, i. e., $y$ consists of a number of zeros, followed by a number of ones. The above argumentation does not work anymore, because $y$ might consist of the same number of ones and zeros. But pumping would still not work: The word $xyyz$ would start out with zeros, then have some ones followed by zeros, and then some more ones. The numbers of ones and zeros might be identical, but the structure of $xyyz$ violates the language rules.

Since none of the ways to split $w$ can be pumped, the language violates the Pumping Lemma and can therefore not be regular.

Sometimes the second condition in the Pumping Lemma can be used to simplify these proofs: If we make the word $w$ much longer than $p$, say picking $w = \mathtt{0}^p\mathtt{1}^p$, we know from the second condition that $y$ consists only of zeros – $|xy| \leq p$, and the first $p$ characters of $w$ are all zeros. By using this special word, we only have to consider the first case in the above proof. Picking the right word to apply the Pumping Lemma to is an art form.

## 2.2 The Language of Regular Expressions Over an Alphabet $\Sigma$

We will now look at the language of regular expressions over an alphabet $\Sigma$ and show that it is not regular. This example is a case where it is absolutely crucial to pick the correct word $w$ for pumping. As in the last example, we start by assuming the language is regular, supplying us with a pumping length $p$. We will apply the pumping lemma to the following word: Let $a \in \Sigma$ be any character, and let $w$ be the regular expression that defines the language $\{a^{p+1}\}$ by concatenating the character $a$ with itself $p + 1$ times. Fully parenthesized, the regular expression looks like the following: $(\cdots((aa)a)\cdots a)$. It is clear that we chose $w$ carefully to start out with exactly $p$ opening parentheses. Now, from the Pumping Lemma it follows that $w$ can be split into $w = xyz$, where $|xy| \leq p$. Since $w$ starts with $p$ opening parentheses, $y$ can only consist of opening parentheses. Therefore, no matter how often we pump $y$, the resulting word $w'$ will have more opening than closing parentheses, and thus cannot be a valid regular expression. Therefore, the Pumping Lemma does not apply, and the language of regular expressions cannot be regular.

# 3 Using the Pumping Lemma Indirectly

Sometimes, a language that is suspected to be non-regular does not lend itself easily to application of the Pumping Lemma. This is typically the case for languages where the words do not have a simple structure. For the language $L_1 := \left\{ \mathtt{0}^n\mathtt{1}^n \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; n \geq 0 \right\}$, all words follow a simple pattern (all zeros followed by all ones), and constructing a contradiction is easy. The language $L_2 := \left\{ w \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; w \text{ has an equal number of zeros and ones} \right\}$ seems more tricky, because the words in $L_2$ can be any mixture of zeros and ones and lack a clear structure.

To attack problems like this, it is often helpful not to apply the Pumping Lemma directly to the language in question, but to use knowledge about the closure properties of regular languages instead. For example, we know that if $A, B \subset \Sigma^*$ are both regular languages over an alphabet $\Sigma$, then the following operations also generate regular languages:

- The complement $\overline{A} = \Sigma^* \setminus A$,
- the union $A \cup B$, the intersection $A \cap B$ and the concatenation $AB$,
- the set difference $A \setminus B = A \cap \overline{B}$, and
- the Kleene Star $A^*$.

Now we can use the following reasoning, for example: If $A$ is the language in question, and $B$ is a language known to be regular, then the result of $A \cap B$ is regular if $A$ is regular itself. In symbols, this statement reads $A$ regular $\wedge$ $B$ regular $\implies A \cap B$ regular. When we negate both sides of the statement, the implication arrow turns around, leaving us with $A \cap B$ not regular $\implies$ $A$ not regular $\vee$ $B$ not regular. When we know that $B$ is in fact regular, the statement simplifies to $A \cap B$ not regular $\implies A$ not regular. This means, if $A \cap B$ is a language that can easily be proven not regular, then $A$ itself cannot be regular.

For example, let us apply this reasoning to the language $L_2$ from above. If we intersect this language with the regular language $L_3 := L\big(\mathtt{0}^*\mathtt{1}^*\big)$, we get exactly the language $L_1$: The words in $L_2 \cap L_3$ must have equal numbers of zeros and ones, and can have only one block of zeros followed by one block of ones. But we have already proven that $L_1$ is nonregular; therefore, $L_2$ must be nonregular as well.