TDDD14/TDDD85 Lecture 6: Pumping Lemma, Myhill-Nerode, and Homomorphisms

Victor Lagerkvist (based on slides by Christer Bäckström)

So far in the course we have mainly focused on providing methods to prove regularity of languages. In this lecture we turn to the opposite problem: how can we prove that a given language is *not* regular? We describe two methods for this purpose, the pumping lemma and the Myhill-Nerode theorem, and consider a new closure property of regular languages which in some circumstances can also be used to prove non-regularity.

Background and Intuition

By now we have seen several characterizations of regular languages and have a vast array of tools to prove that a language is regular. For example, given a language A we could: construct a DFA, construct an NFA, construct a regular expression, or show that A can be expressed as the union, star, or concatenation, of simpler, regular languages. But what if all these methods fail? In fact, what if they cannot succeed, since the language in question is *not* regular?

The most straightforward example of a non-regular language stems from the following application: assume that we in the context of parsing of programming languages need the ability to recognise whether a sequence of left parentheses (and right parentheses) is evenly balanced, in the sense that each left parenthesis is closed by a matching right parenthesis. In fact, let us make this scenario even simpler: given a string x we merely want to be able to decide whether it can be broken down into two parts $x = x_1x_2$ where x_1 consists of n consecutive left parentheses, and x_2 of n consecutive right parentheses (for some n). To make this typographically simpler we furthermore write 0 to indicate a left parenthesis (, and 1 to denote a right parenthesis). Hence, the language in question is $\{0^n1^n \mid n \geq 1\}$, meaning that 000111 is included but 00011 is not. Might it be possible to construct a DFA for this language? This would be good news since DFAs can be implemented efficiently with a low memory footprint.

However, it is far from obvious that this is possible. It is easy to see that we for each *fixed* k directly can accept the string 0^k1^k by "hardwiring" a DFA with $2k + 1$ states which accepts 0^k1^k and nothing else. More generally, if we have a DFA with n states which claims to accept L then it seems that we could "trick" it simply by feeding it the string $0^{n+1}1^{n+1}$, which would force it to return to a previously visited state and forget the number of zeroes that it has read. Com-

pare this with the natural strategy of recognising this language in a computer program, e.g., by (1) incrementing a counter each time we read 0, and (2) decrement the counter each time we read 1, and after having exhausted the input string we accept the string if and only if the total count is 0. However, this strategy cannot be encoded by a DFA since this counter cannot be of a fixed size.

Hence, we suspect that $\{0^n 1^n \mid n \geq 1\}$ is not regular, but while it is good to have an intuition of why this might be the case, a formal proof would be even better. But how can we rule out that *every* DFA fails to recognise the language? We will present two methods for this purpose, the *pumping lemma* and the *Myhill-Nerode theorem*, and in doing so we will also establish important properties of regular languages.

The Pumping Lemma for Regular Languages

We begin by describing an important property of regular languages and finite automata called the *pumping lemma*. The basic idea is simple: if we have a DFA and take a sufficiently long input string and simulate the DFA then some states will be visited several times since each DFA has a fixed number of states. More precisely, let M be a DFA with n states. Suppose M accepts some string $s_1, s_2, \ldots, s_n$ of length n. Then there must be $n+1$ states $r_0, r_1, \ldots, r_n$ such that[1]

[1] See Definition 3 in Lecture 2 if you are confused why the number of states must be $n+1$ in this case.

$$r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} r_2 \ldots \xrightarrow{s_n} r_n.$$

However, M has only n states, so there must be some k and l such that $r_k = r_l$. Without loss of generality, assume that $k < l$ (Figure 1).

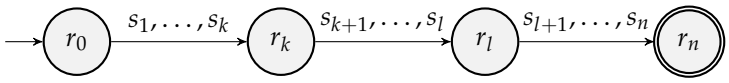

Figure 1: After having read $s_1, \ldots, s_k, s_{k+1}, \ldots, s_l$ the machine returns to $r_k = r_l$.

Hence, the machine returns to r_k (Figure 2).

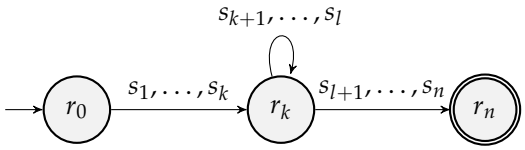

Figure 2: Since $r_k = r_l$ the sequence of transitions can be visualised as a loop.

But if we previously had the choice of going from $r_l = r_k$ to r_n by reading $s_{l+1}, \ldots, s_n$ then it must be possible to go directly from r_k to r_n by reading $s_{l+1}, \ldots, s_n$ (Figure 3).

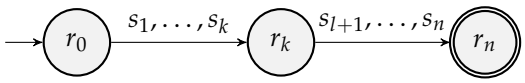

Figure 3: M must also accept $s_0, \ldots, s_k, s_{l+1}, \ldots, s_n$.

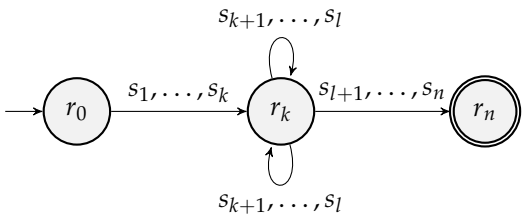

Figure 4: M must also accept $s_0, \ldots, s_k, (s_{k+1}, \ldots, s_l)^2, s_{l+1}, \ldots, s_n$.

In addition, since we previously went from r_k to r_k by reading $s_{k+1}, \ldots, s_l$ there is nothing that prevents us from repeating this one additional time (Figure 4).

Naturally, we can repeat this a third time, a fourth time, and so on, as long as we desire. Thus, for each $i \geq 0$ the automaton M accepts the string $s_0, \ldots, s_k, (s_{k+1}, \ldots, s_l)^i, s_{l+1}, \ldots, s_n$. Constructing the string $s_0, \ldots, s_k, (s_{k+1}, \ldots, s_l)^i, s_{l+1}, \ldots, s_n$ is called "pumping", and this property of *all* regular languages can be summarised as follows.

Lemma 1. *(Pumping lemma) If L is a regular language, then there exists a positive integer p (the pumping length) such that every string $s \in L$, where $|s| \geq p$, can be partitioned into three pieces, $s = xyz$, such that the following conditions hold:*

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

Proof. (Sketch)

If L is regular, then there is some DFA M that recognizes L. Let p be the number of states of M, and let $s = s_1, s_2, \ldots, s_n$ be a string in $L(M)$ such that $n \geq p$.

Let $r_0, r_1, \ldots, r_n$ be the states M passes when reading s, where r_0 is the start state and r_n is an accept state. It follows that two of the first $p + 1$ states must be the same, i.e., there exists k and l $(0 \leq k < l \leq p)$ such that $r_k = r_l$. We partition s as

$$s = \underbrace{s_1, \ldots, s_k}_{x}, \underbrace{s_{k+1}, \ldots, s_l}_{y}, \underbrace{s_{l+1}, \ldots, s_n}_{z}.$$

See Figure 5 for a visualisation. Since $r_k = r_l$ we can repeat y any number of times, including 0 (Figure 6). Hence, M accepts all strings of the form xy^iz for all $i \geq 0$. Put together, this gives us

1. $|y| > 0$, since $y = s_{k+1}, \ldots, s_l$ and $k < l$,

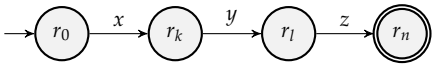

Figure 5: The DFA transitions from r_0 to r_n by reading $s = xyz$ where $r_k = r_l$.

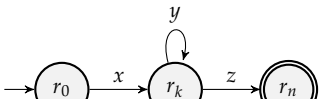

Figure 6: Since $r_k = r_l$ we can loop in r_k.

2. $|xy| \leq p$, since $k, l \leq p$,
3. $xy^iz \in L(M)$ for all $i \geq 0$,

and have thus proven the pumping lemma. □

In the next section we will see that the pumping lemma is a useful tool to prove that a given language is *not* regular.

Inverting the Pumping Lemma

The pumping lemma proves a property of regular languages: *if* a language is regular *then* the conditions in the pumping lemma hold. If we want to prove that a language is *not* regular, then we must "invert" the lemma. Hence, if we let R be the statement "L is regular" and P be the rest of the statement "there exists a positive integer $p, \ldots$" from the pumping lemma then the pumping lemma states: $R \Rightarrow P$. However, this is logically equivalent to the contrapositive form: $\neg P \Rightarrow \neg R$. Hence, if we can prove that $\neg P$ is true, i.e., that P does *not* hold, then we would obtain a method for proving non-regularity. We provide a detailed exposition of how to simplify $\neg P$ in the appendix, and for the moment simply state the resulting variant of the pumping lemma as follows.

Lemma 2. *(Inverted pumping lemma) If there for each positive integer p (the pumping length) exists a string $s \in L$, $|s| \geq p$, such that for each partitioning $s = xyz$ where*

1. $|y| > 0$,
2. $|xy| \leq p$, *and*
3. *there exists $i \geq 0$ such that $xy^iz \notin L$,*

then L is not regular .

To reiterate, the inverted pumping lemma implies that to prove that a language L is not regular, we must:

1. Assume an arbitrary pumping length p (*we cannot choose it*).
2. Choose a suitable string $s \in L$ of length $\geq p$.
3. Show that for *all possible choices* of strings x, y, z such that $s = xyz$, $|y| > 0$ and $|xy| \leq p$, there is some $i \geq 0$ s.t. $xy^iz \notin L$.

Why then, do we state and prove the pumping lemma in the form of Lemma 1, and not its inverted form? The reason is simply that we in mathematics typically strive for simplicity. Hence, instead of proving a statement $\neg B \Rightarrow \neg A$ we prefer the simpler, direct statement $A \Rightarrow B$. Let us now see how the pumping lemma can be used to prove non-regularity of a language.

Example 1. *We will use the (inverted) pumping lemma to prove that* $L = \{0^n1^n \mid n \geq 0\}$ *is not regular. The basic idea is to produce a string which is not of the form* 0^n1^n *by finding a suitable string and a segment in this string which we can "pump" in order to create a string with more zeroes than ones. Hence, the idea is rather straightforward, but we need to be careful to ensure that we strictly follow the premises.*

1. *Assume L has a pumping length p. Hence, we cannot make any assumptions on p, we just know that it is an arbitrary positive natural number.*
2. *Next, choose* $s = 0^p1^p$*, which is in L. Note that for the particular language L we did not have any meaningful choice of the string s, since each string in L is of the form* 0^n1^n *for some* $n \geq 1$*.*

For all *choices of* x, y, z *such that* $s = xyz$*,* $|y| > 0$ *and* $|xy| \leq p$*, the string xy can only contain zeroes*[2]*. Hence, it must hold that:*

1. $y = 0^m$ *for some* $m > 0$*,*
2. $x = 0^k$ *for some* $k \geq 0$ *s.t.* $k + m \leq p$ *and*
3. $z = 0^{p-k-m}1^p$*.*

Note that the constraints on k and m cover all possible choices of x, y and z, and we have $xyz = 0^k0^m0^{p-k-m}1^p = 0^p1^p = s$*. Last, to prove the claim we must show that there exists an* $i \geq 0$ *such that* $xy^iz \notin L$*. However, this is very simple for the language L. Simply choose* $i = 2$*. We get* $xy^2z = 0^k0^{2m}0^{p-k-m}1^p = 0^{p+m}1^p \notin L$*. It follows that L cannot be regular.*

[2] Since $s = 0^p1^p$ any prefix of length $\leq p$ can only consist of zeroes
Hint: to get an intuition of the partitioning $s = xyz$ it can be worthwhile to consider a few concrete examples of $s \in L$. These examples are *only* for illustrative purposes and should *not* be part of your written solution.

From this example we see the most difficult part of applying the pumping lemma is to partition the string s into xyz of the required form. However, once this is done, it is typically rather easy to finish the proof by producing a string not in the language. Last, let us remark that the pumping lemma as we have proved it only gives a necessary, but not sufficient, condition for regularity. Hence, there are languages whose non-regularity cannot be proven by the pumping lemma, but which *can* be proved non-regular by the method proposed in the forthcoming section.

The Myhill-Nerode Theorem

We now turn to a second method for proving non-regularity. This method is more powerful than the pumping lemma and leads to an entirely new, complete, characterization of regular languages. However, the theorem is more complicated than the pumping lemma and harder to visualize, so we begin by stating it in an informal way.

Theorem 1. *(Informal Myhill-Nerode) Let $L \subseteq \Sigma^*$ be a language. Then the following statements are equivalent:*

1. *L is regular,*
2. *there exists a special equivalence relation over Σ^* describing L[3],*
3. *there exists a special equivalence relation encoding a unique minimal automaton recognising L.*

[3] Roughly: each DFA for L can be described by such an equivalence relation, and for each such equivalence relation one can construct a DFA for L.

The third item is the most relevant for our purposes so we begin by describing this "special equivalence relation".

Definition 1. *Let Σ be an alphabet and let $L \subseteq \Sigma^*$ be a language (L need not be regular.) Define the relation $\equiv_L$ such that for all $x, y \in \Sigma^*$ $x \equiv_L y$ if and only if for all $z \in \Sigma^*$, $xz \in L \Leftrightarrow yz \in L$.*

Thus, two strings are equivalent under $\equiv_L$ whenever appending a third string does not affect membership in the language L. It is not so difficult to prove that $\equiv_L$ is an equivalence relation over Σ^*, and the idea behind the definition of $\equiv_L$ is then that its equivalence classes correspond to states in a minimal DFA recognising L[4]. In addition, $\equiv_L$ is known to

1. be *right congruent*, i.e., for all $x, y \in \Sigma^*$ and all $a \in \Sigma$, if $x \equiv_L y$, then $xa \equiv_L ya$, and
2. *refine* L, i.e. if $x \equiv_L y$, then $x \in L \Leftrightarrow y \in L$.

[4] Recall that in the minimisation algorithm we collapsed states if they led to the same outcome with respect to acceptence and rejectence. In other words, take two states p and q that are equivalent, and assume that the machine has read x when in state p and y when in state q. Then, for any string z, the machine either accepts both xz and yz or has to reject both xz and yz. The intuition behind $\equiv_L$ is then that if we take two strings x and y in the same equivalence class, then for any third string z, either xz and yz are both in the language, or are both not in the language. Hence, $\equiv_L$ can be seen as an encoding of a minimal DFA for L, but can be defined directly from the language L and does not have to be constructed from a specific DFA.

Depending on the language L then $\equiv_L$ might or might not be of *finite index*, meaning that $\equiv_L$ has a finite number of equivalence classes.

Example 2. *Let us return to the language $L = \{x1 \mid x \in \{0,1\}^*\}$ which we constructed a minimal automaton for in the previous lecture, and see what the corresponding relation $\equiv_L$ looks like. Then we e.g. have that*

1. $0 \equiv_L \varepsilon$,
2. $0 \equiv_L 0$,
3. $1 \equiv_L 11$,
4. $0 \equiv_L 00$,
5. $0^n \equiv_L 0^m$ *for all* $n, m \geq 1$,
6. $01 \equiv_L 11$,
7. $x1 \equiv_L y1$ *for all* $x, y \in \{0,1\}^*$.

However, 01 *is* not *related to* 00 *since if we let* $z = \varepsilon$ *then* $01\varepsilon = 01 \in L$ *but* $00\varepsilon = 00 \notin L$*. Based on these examples we see that* $\equiv_L$ *only has the two equivalence classes* $[0] = \{x0 \mid x \in \{0,1\}^*\} \cup \{\varepsilon\}$ *and* $[1] = \{x1 \mid x \in \{0,1\}^*\}$*, which exactly corresponds to the states of the minimal automaton that we constructed in the previous lecture.*

In fact, *if* the relation $\equiv_L$ is of finite index, then these equivalence classes exactly correspond to the number of states in a minimal DFA accepting L. Similarly, the Myhill-Nerode theorem states that *if* L is regular, then $\equiv_L$ is of finite index and there exists a unique minimal DFA accepting L[5]. Moreover, this DFA is precisely the minimal DFA constructed by the minimisation algorithm in the previous lecture.

However, the Myhill-Nerode theorem is even stronger than this, but before we can relate the equivalence relation $\equiv_L$ to the second item in the Myhill-Nerode theorem we have to properly define this statement.

Definition 2. *Let* Σ *be an alphabet and let* $L \subseteq \Sigma^*$ *be a language. An equivalence relation* $\equiv$ *on* Σ^* *is a a* Myhill-Nerode relation *for* L *if it is right congruent, refines* L*, and is of finite index.*

As we have already stated, $\equiv_L$ is not necessarily a Myhill-Nerode relation since it might not be of finite index. The idea behind Myhill-Nerode relations is then that one for each DFA can "encode" the automaton by defining a suitable Myhill-Nerode relation.

Example 3. *For each DFA* M *there exists a canonical Myhill-Nerode relation which can be defined as follows. Given a DFA* M *we define the relation* $\equiv_M$ *such that*

$$x \equiv_M y$$

if and only if

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$

for any $x, y \in \Sigma^*$[6]*. Then* $\equiv_M$ *is an equivalence relation*[7] *over* Σ^* *where any two strings are "equivalent" if they lead to the same state in the automaton* M*. We will soon also see that* M *is a Myhill-Nerode relation for* $L(M)$*.*

Note that $\equiv_M$ is defined with respect to a given *DFA* M, while the relation $\equiv_L$ is defined with respect to the given *language* L. The Myhill-Nerode theorem then relates these concepts together as follows.

Theorem 2. *(Myhill-Nerode) Let* $L \subseteq \Sigma^*$ *be a language. Then the following statements are equivalent:*

1. *L is regular,*
2. *there exists a Myhill-Nerode relation for L,*

[5] At least up to renaming of states, i.e., up to isomorphism.

[6] Recall from the previous lecture that $\hat{\delta}$ is the extension of δ to strings over Σ^*.

[7] Why? It is a good exercise to attempt to prove this.

3. *the relation $\equiv_L$ has a finite number of equivalence classes.*

We provide a short proof sketch of the main ideas in the appendix. For now, it is sufficient to understand (1) that Myhill-Nerode relations are equivalence relations corresponding to DFAs in a very precise way, and (2) that (non)regularity of a language can always be proven by showing that $\equiv_L$ has a (in)finite number of equivalence classes. Furthermore, the number of equivalence classes of $\equiv_L$ exactly corresponds to the number of states in a minimal DFA accepting L. Hence, we have developed an entirely new method to prove that a language is regular by using equivalence relations which (at least superficially) seems to be quite far away from automata. Let us now consider a "practical" application of the Myhill-Nerode theorem which we can use to prove non-regularity of languages.

Example 4. *We return to the language $L = \{0^n 1^n \mid n \geq 1\}$, which we have already proven to be non-regular by the pumping lemma, and give an alternative and, arguably, more elegant proof using the Myhill-Nerode theorem. For each $0^n \in \{0,1\}^*$, $n \geq 1$, consider the equivalence class $[0^n]$. Could it be the case that $0^m \in [0^n]$ for any $m \neq n$? If this were the case, then $0^n \equiv_L 0^m$, and by definition of $\equiv_L$ it would then hold that $0^n z \in L$ if and only if $0^m z \in L$ for* any *string $z \in \{0,1\}^*$. However, this is evidently false since $0^n 1^n \in L$ but $0^m 1^n \notin L$. Hence, $\equiv_L$ has an infinite number of equivalence classes, which implies that L is not regular by the Myhill-Nerode theorem.*

Importantly, this example shows that the Myhill-Nerode theorem can give a much more succinct proof of non-regularity, and it can do so without us needing to remember exactly what a Myhill-Nerode relation is, since we for each language L only need to take the definition of $\equiv_L$ and determine the number of equivalence classes.

Pumping Lemma versus Myhill-Nerode

Hence, we have two methods to prove non-regularity of languages. Which one is preferable to use in which case, and why do we present both? For example, the Myhill-Nerode has the following advantages.

1. Gives a *complete* condition for regularity: if a language L is regular then $\equiv_L$ has finite index, otherwise not.
2. Can result in more succinct proofs.
3. Proves that each regular language has a unique, minimal DFA up to isomorphism.
4. Less weird name.

On the other hand, the pumping lemma also has several advantages.

1. More direct argument which is directly related to how a DFA operaters (for sufficiently long strings the DFA will loop on some states).
2. We will see a generalisation which is applicable to *context-free languages*.
3. Much easier to prove.

In the end, there are advantages and disadvantages to both, but since the Myhill-Nerode theorem does not generalise to context-free languages we have a larger focus on the pumping lemma in this course.

Homomorphisms

In this section we describe a closure property of regular languages which we have previously omitted. This new closure property, closure under *homomorphisms*, can sometimes be useful for proving that a language is not regular given that we already know a non-regular language (e.g., $\{0^n1^n \mid n \geq 1\}$).

Definition 3. *Let Σ and Γ be two alphabets. A function $h\colon \Sigma^* \to \Gamma^*$ is a* homomorphism[8] *if*

1. $h(xy) = h(x)h(y)$ *for all* $x, y \in \Sigma^*$
2. $h(\varepsilon) = \varepsilon$.

[8] homo ≈ similar, morphism ≈ shape. Hence, think of a homomorphism as a mapping resulting in a similar shape. In this case concatenation has a "similar shape" after the homomorphism.

Note that *any* function h from Σ (i.e., the alphabet) to Δ^* results in a homomorphism from Σ^* to Δ^*, since we can extend h to strings over Σ^* in the natural way by letting $h(s_1s_2\dots s_n) = h(s_1)h(s_2)\dots h(s_n)$ and $h(\varepsilon) = \varepsilon$. In fact, every homomorphism between two languages can be described in this way.

Example 5. *Let $\Sigma = \Gamma = \{a, b, c\}$ and define h as*

$$h(a) = bab, h(b) = cbc, h(c) = a.$$

Then $h(abc) = h(a)h(b)h(c) = babcbca$.

Homomorphisms can be extended to sets in a straightforward way, i.e., if $A \subseteq \Sigma^*$, then $h(A) = \{h(x) \mid x \in A\}$. The set $h(A)$ is the *image* of A under h (see Figure 7).

It can then be proven that $h(A)$ is regular whenever A is regular.

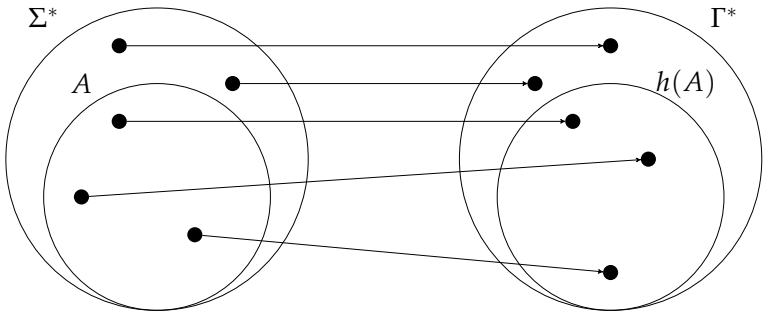

Figure 7: A visualisation of a homomorphism between Σ^* and Δ^*.

Theorem 3. *If A is regular, then $h(A)$ is regular.*

This implies that if $h(A)$ is *not* regular then A cannot be regular, either. Hence, if we already have a language which is not regular and manage to provide a homomorphism to this language, then the original language cannot be regular, and we do not have to prove non-regularity from scratch by using the pumping lemma or the Myhill-Nerode theorem. For a simple example, consider the language $A = \{a^n b^n c \mid n \geq 1\}$ over the alphabet $\{a, b, c\}$. We strongly suspect that this language is not regular since it is very similar to $\{0^n 1^n \mid n \geq 1\}$, and to prove this we can define the homomorphism $h(a) = 0$, $h(b) = 1$, $h(c) = \varepsilon$, since then $h(A) = \{0^n 1^n \mid n \geq 1\}$.

We will not attempt to prove Theorem 3 (see, e.g., Theorem 10.2 in Kozen [9]) and instead consider an example which highlights the main idea.

[9] D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997

Example 6. *Let us try to get some intuition for why $h(A)$ is regular if A is regular. Let $\Sigma = \{0, 1, 2, 3\}$ and let $\Gamma = \{0, 1\}$. Define h as a binary encoding of Σ, i.e., $h(0) = 00$, $h(1) = 01$, $h(2) = 10$ and $h(3) = 11$. We now proceed as usual: if $A \subseteq \Sigma^*$ is regular, then there is some DFA M for A, and we need to construct a DFA for $h(A)$ using M. Assume that we have a state s and transitions 2 and 3 in the machine M (Figure 8). Then $h(2) = 10$ and $h(3) = 11$, so we add a new state q' to the new DFA (Figure 9) such that we end up in r if we read 10 from q, and in s if we read 11.*

Analogously, one then introduces new states for all other pairs of transitions. This results in a DFA and it is not so difficult that it recognises $h(A)$.

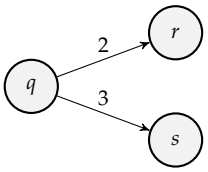

Figure 8: q, r, s in the DFA M.

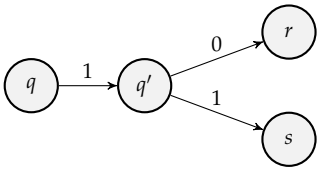

Figure 9: We add a new state q' in the DFA for $h(A)$.

We also define the backwards direction: if $B \subseteq \Gamma^*$, then

$$h^{-1}(B) = \{x \in \Sigma^* \mid h(x) \in B\}.$$

The set $h^{-1}(B)$ is the *preimage* of B under h and is visualized in Figure 10. Similarly to the previous case, one can prove that $h^{-1}(B)$ is regular if B is regular.

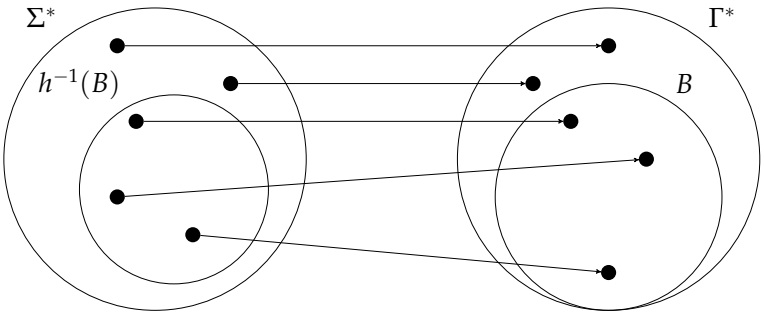

Figure 10: A visualisation of the preimage of B under h.

Summary

We presented two methods for proving non-regularity of languages: the pumping lemma and the Myhill-Nerode theorem. In addition we saw that regular languages are closed under homomorphisms, providing another tool for proving non-regularity.

Food for Thought

1. We have proved that $\{0^n1^n \mid n \geq 1\}$ is not regular. Write a function in your favourite programming language which recognises this language, e.g., by returning 1 if the input string is in the language, and 0 otherwise. For how large n does your program work? How much memory does your program need with respect to the length of the input string? What happens (in principle) if the input string is of length 2^k for some large k? Will your computer run out of memory? If so, is your computer really more powerful than a DFA?

2. Assume a person puts forth the following **incorrect** argument for proving that $L = \{0^n1^n \mid n \geq 1\}$ is not regular.

   (a) I will prove that L is not regular by finding a string in the language which can be pumped so that the resulting string is not in L.

   (b) Hence, define the string $s = 000111$ where the pumping length $p \leq 3$.

   (c) Then, for any partitioning $s = xyz$ where $|xy| \leq p$ and every $i \geq 0$ the string xy^iz is not in L.

   (d) Hence, L is not regular.

   What is/are the error(s) in the above claims?

3. Assume that the person in the previous question is very stubborn and now claims to have yet another proof (sadly, also **incorrect**) of non-regularity of L.

   (a) I will prove that L is not regular by finding a string in the language which can be pumped so that the resulting string is not in L.

   (b) Hence, let $p > 0$ and pick a string s from $\{0^n 1^n \mid n \geq 1\}$ of length at least p.

   (c) Then, partition s into xyz where $|xy| \leq p$ and $xy = 0^n$. Then, for every $i \geq 2$ the string $xy^i z$ is not in L.

   (d) Hence, L is not regular.

   What is/are the error(s) in the above claims?

Appendix

The Inverted Pumping Lemma

We begin by rewriting the pumping lemma using logic notation:
L regular $\rightarrow$

$$\exists p > 0 \, \forall s \in L(|s| \geq p) \, \exists x, y, z \, . \, (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \geq 0 \, . \, xy^i z \in L).$$

We will use the following rules of logic:

1. $\neg \forall x \, . \, \phi(x) \Leftrightarrow \exists x \, . \, \neg \phi(x)$,
2. $\neg \exists x \, . \, \phi(x) \Leftrightarrow \forall x \, . \, \neg \phi(x)$,
3. $\neg(\phi_1 \wedge \phi_2 \wedge \ldots, \wedge \phi_n) \Leftrightarrow (\neg \phi_1 \vee \neg \phi_2 \vee \cdots \vee \neg \phi_n)$ (De Morgan),
4. $(\phi \vee \psi) \Leftrightarrow (\neg \phi \rightarrow \psi)$,
5. $(\phi_1 \vee \cdots \vee \phi_{n-1} \vee \phi_n) \Leftrightarrow ((\neg \phi_1 \wedge \cdots \wedge \neg \phi_{n-1}) \rightarrow \phi_n)$ $(3 + 4)$, and
6. $(\phi \rightarrow \psi) \Leftrightarrow (\neg \psi \rightarrow \neg \phi)$.

We begin by using rule 6 and rewrite the statement into its contrapositive form:
L not regular $\leftarrow$

$$\neg \exists p > 0 \, \forall s \in L(|s| \geq p) \, \exists x, y, z \, . \, (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \geq 0 \, . \, xy^i z \in L).$$

This expression is equivalent to the pumping lemma, but the right-hand-side of the implication now states a condition for when L is *not*

regular. We now want to rewrite the right-hand-side to a more useful form by simplifying the expression as much as possible.

Use rule 2:

$$\forall p > 0\, \neg\forall s \in L(|s| \geq p)\, \exists x, y, z\,.\, (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \geq 0\,.\, xy^i z \in L.$$

Use rule 1:

$$\forall p > 0\, \exists s \in L(|s| \geq p)\, \neg\exists x, y, z\,.\, (s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \geq 0\,.\, xy^i z \in L).$$

Use rule 2:

$$\forall p > 0\, \exists s \in L(|s| \geq p)\, \forall x, y, z\,.\, \neg(s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge \forall i \geq 0\,.\, xy^i z \in L).$$

Use rule 3 (De Morgan)

$$\forall p > 0\, \exists s \in L(|s| \geq p)\, \forall x, y, z\,.\, (s \neq xyz \vee |y| \not> 0 \vee |xy| > p \vee \neg\forall i \geq 0\,.\, xy^i z \in L).$$

Use rule 1 on the innermost quantifier:

$$\forall p > 0\, \exists s \in L(|s| \geq p)\, \forall x, y, z\,.\, (s \neq xyz \vee |y| \not> 0 \vee |xy| > p \vee \exists i \geq 0\,.\, xy^i z \notin L).$$

Finally, turn the disjunction into an implication, using rule 5:

$$\forall p > 0\, \exists s \in L(|s| \geq p)\, \forall x, y, z\,.\, ((s = xyz \wedge |y| > 0 \wedge |xy| \leq p) \rightarrow \exists i \geq 0\,.\, xy^i z \notin L).$$

See Lemma 2 for the resulting variant of the pumping lemma.

The Myhill-Nerode Theorem

In this section we give a short proof sketch of the Myhill-Nerode theorem by proving item 2 and item 3 in turn. For a detailed proof we refer the interested reader to Kozen [10].

[10] D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997

Lemma 3. *A language L is regular if and only if there exists a Myhill-Nerode relation for L.*

Proof. (Sketch) Let M be a DFA recognising L. We begin by proving that the equivalence relation $\equiv_M$ from Example 3 is a Myhill-Nerode relation for L. For example, to prove that $\equiv_M$ is right congruent we may argue as follows. Let x and y be arbitrary strings in Σ^* and let a be an arbitrary symbol in Σ. Assume $x \equiv_M y$. Then $\hat{\delta}(q_0, x) =$

$\hat{\delta}(q_0, y)$ by definition. But then we get $\hat{\delta}(q_0, xa) = \delta(\hat{\delta}(q_0, x), a) = \delta(\hat{\delta}(q_0, y), a) = \hat{\delta}(q_0, ya)$. That $\equiv_M$ refines L and that it is of finite index can also proven with straightforward arguments.

For the other direction, assume that $\equiv$ is a Myhill-Nerode relation for L. The basic idea is then, since $\equiv$ is an equivalence relation, to define an automaton over the set of states $\{[x] \mid x \in \Sigma^*\}$, which is *finite* since $\equiv$ is a Myhill-Nerode relation, and must therefore be of finite index. The start state is simply $[\varepsilon]$, the set of final states is $\{[x] \mid x \in L\}$, and the transition function δ is defined as $\delta([x], a) = [xa]$. It can then be proven that the resulting automaton recognises L. □

The construction in the proof of Lemma 3 is in fact even more powerful: if L is a regular language with a Myhill-Nerode relation $\equiv$ and if we construct the DFA $M_\equiv$ corresponding to $\equiv$, then the Myhill-Nerode relation $\equiv_{M_\equiv}$ constructed from $M_\equiv$ turns out to coincide with $\equiv$[11]. Similarly, it is possible to prove that if one begins with a DFA M and constructs the Myhill-Nerode relation $\equiv_M$, then the DFA $M_{\equiv_M}$ constructed from $\equiv_M$ coincides with M.

[11] Up to isomorphism.

Lemma 4. *Let L be a language. Then there exists a Myhill-Nerode relation for L if and only if $\equiv_L$ has a finite number of equivalence classes.*

Proof. If $\equiv$ is a Myhill-Nerode relation for L then $\equiv$ has a finite number of equivalence classes. It is then known that $\equiv_L$ is *coarser* than $\equiv$, meaning that $\equiv \subseteq \equiv_L$. Since $\equiv$ and $\equiv_L$ are both equivalence relations this means, informally that $\equiv_L$ relates more elements together and therefore have fewer (or as many) equivalence classes as $\equiv$. Hence, $\equiv_L$ has a finite number of equivalence classes since $\equiv$ has a finite number of equivalence classes.

The second statement follows directly by the given assumptions: $\equiv_L$ is already known to be right congruent and to refine L, and if it has a finite number of equivalence classes then it must be a Myhill-Nerode relation. □

References

D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997.