Chapter 2

Regular Languages

In the introductory chapter, we have seen how we can define languages in terms of grammars using production rules. However, phrase structure grammars can be too general for certain applications. For example, it is not clear how to write a program which decides whether a given string is in the language generated by a grammar. We will start by looking at restricted grammars and then slowly allow more and more complex ones until we can discuss general phrase structure grammars.

2.1 Regular Grammars

The first class of restricted grammars are so called *regular grammars.* The idea it that we would like to be able to scan a string from left to right, deciding whether or not it is in the language in a straightforward manner.

Definition: A *regular grammar* is a phrase-structure grammar $\langle \Sigma, N, I, P \rangle$ such that every production rule is in the form $A \to aB$, $A \to a$ or $A \to \varepsilon$ (where $A, B \in N$, $a \in \Sigma$). In other words, $P \subseteq N \times (\{\varepsilon\} \cup \Sigma N \cup \Sigma)$.

Note that with this constraint, we have at most one non-terminal symbol in the string during a derivation, which simplifies things considerably. Furthermore, the non-terminal always appears at the end of the string.

We say that a regular grammar G is *non-deterministic* if there are two rules of the form $A \to aB$ and $A \to aC$ (for the same A and a) or two rules $A \to a$ and $A \to aB$ (for the same a and A). A regular grammar which is not non-deterministic, is obviously called *deterministic.*

Given a string $s \in \Sigma^*$ and deterministic grammar G, it is straightforward to check whether $s \in \mathcal{L}(G)$. The idea is simply to scan string s left to right, one symbol at a time, and choose the corresponding production depending on the current non-terminal and next symbol in s. Note that thanks to determinism, we never have more than one choice we can make. In fact, it is not much

more complicated to generalise this algorithm to work for non-deterministic grammars. However, we will see that there is a much more straightforward way of answering the membership question in the next section.

Definition: A language L is said to be regular, if there exists a regular grammar G which generates L: $L = \mathcal{L}(G)$.

We have thus characterised a class of languages based on the grammar that can generate it. We will later show that not all languages are regular.

Definition: A phrase structure grammar $G = \langle \Sigma, N, I, P \rangle$ is called an *extended regular grammar* if all production rules are of the form $N \times (N \cup \Sigma \cup \Sigma N \cup \{\varepsilon\})$.

In other words, we have also permitted rules of the form $A \to B$. In fact, every language described by an extended regular grammar can also be generated by a standard regular grammar (see next theorem). However, we will be able to construct certain grammars more easily using these extra rules.

Theorem: Extended regular grammars are just as expressive as regular grammars.

Proof: We would like to prove that if $G = \langle \Sigma, N, I, P \rangle$ is an extended regular grammar, then there exists a regular grammar $G' = \langle \Sigma, N', I', P' \rangle$ such that $\mathcal{L}(G) = \mathcal{L}(G')$.

Given a set of extended regular grammar production rules P, we will define $\overline{P}$ to be the 'good' productions $P \backslash \{A \to B \mid A \to B \in P\}$.

Furthermore, given a non-terminal A, we define $\overline{\varepsilon}(A)$ to be the non-terminals which can be generated from A (with no other symbols): $\{B \in N \mid A \Rrightarrow^* B\}$. This is computable (see exercise 7).

We can now define G' as follows:

$$\begin{aligned} N' &\stackrel{\text{df}}{=} N \\ I' &\stackrel{\text{df}}{=} I \\ P' &\stackrel{\text{df}}{=} \{A \to \alpha \mid \exists B \in \overline{\varepsilon}(A) \cdot B \to \alpha \in \overline{P}\} \end{aligned}$$

In other words, we throw away all 'bad' transitions, but if $A \Rrightarrow^* B$ and $B \to \alpha$, we add $A \to \alpha$.

The proof that this construction works would follow from the fact that every new transition can be simulated in the old system, and vice-versa.

□

2.2 Finite State Automata

In the first chapter, we also used a simple type of automaton in the shape of a diagram to describe languages. We can rather easily formalise these diagrams:

Definition: A *finite state automaton* is a tuple $M = \langle \Sigma, Q, q_0, F, T \rangle$, where Σ is a finite set of symbols (the alphabet of the automaton), Q is a finite set of states

(the 'names' of the circles in the diagram), q_0 is the state (or circle) with an incoming arrow (the initial or start state) $q_0 \in Q$, F is the set of double circled states (the final states) $F \subseteq Q$ and T is the transition relation of the automaton, telling us where to go from a given state, with a given symbol $T \subseteq Q \times \Sigma \times Q$.

As in the case of grammars, we say that a finite state automaton is non-deterministic if, by reading a symbol a at some state q, we have more than one transition we can follow: $\exists q, q', q'' \in Q, a \in \Sigma \cdot (q, a, q') \in T \wedge (q, a, q'') \in T \wedge q' \neq q''$.

We can now define the 'game' we used to determine whether or not a string is in the language recognised by the diagram mathematically.

We say that we can go from state q to state q' accepting symbol a, if (q, a, q') is in the transition relation. We write this as $q \overset{a}{\Rrightarrow}_1 q' \overset{\text{df}}{=} (q, a, q') \in T$.

We can generalise this to work for longer derivations, over any strings:

$$\begin{array}{lcl} q \overset{\varepsilon}{\Rrightarrow} q' & \overset{\text{df}}{=} & q = q' \\ q \overset{as}{\Rrightarrow} q' & \overset{\text{df}}{=} & \exists q'' \in Q \cdot q \overset{a}{\Rrightarrow}_1 q'' \wedge q'' \overset{s}{\Rrightarrow} q' \end{array}$$

Thus, in a finite state automaton M, we write that $q \overset{s}{\Rrightarrow} q'$ if there is a path from q to q' following string s. Note that there be more than one path due to non-determinism, and thus $q \overset{s}{\Rrightarrow} q'$ does not mean that there may not also be some other state q'' such that $q \overset{s}{\Rrightarrow} q''$.

Using this relation, we can now define what the language generated by an automaton is.

Definition: The language generated by automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$ (written $\mathcal{L}(M)$) is a subset of Σ^*. A string is in this language if it can take us from the start state q_0 to some final state: $\mathcal{L}(M) \overset{\text{df}}{=} \{s \in \Sigma^* \mid \exists q_F \in F \cdot q_0 \overset{s}{\Rrightarrow} q_F\}$

As in the case of regular grammars, it is rather straightforward to decide whether a string is accepted (is in the language generated) by a deterministic finite state automaton.

```
state := q0;
while not(end-of-string(s)) and (state != NULL) do
   a := get-next-symbol(s);
   state := next-state(state, a);
done
accepted := state != NULL and element-of(state, final);
```

With non-deterministic automata, it is slightly more complex, but the following theorem gives us a way of testing whether a string is accepted by a non-deterministic automaton using the algorithm for deterministic automata.

Theorem: For every non-deterministic automaton M, there exists a deterministic automaton M' such that $\mathcal{L}(M) = \mathcal{L}(M')$.

In other words, we are showing that non-deterministic automata are not more expressive than deterministic ones.

Construction: The proof of this theorem is a constructive one. Given $M = \langle \Sigma, Q, s_0, F, T \rangle$, we show how to construct deterministic $M' = \langle \Sigma', Q', s_0', F', T' \rangle$ such that both accept the same language. In fact, the proof of equivalence is tedious and not very illuminating, and we will be leaving it out. Consult a textbook if you are interested in seeing how the proof would proceed.

Before anything else, note that the new automaton will accept strings over the same alphabet, and thus $\Sigma' \stackrel{\text{df}}{=} \Sigma$.

The idea behind the construction is the following: In a non-deterministic automaton, we may have a choice of which state to go to. We will encode such a choice by enriching the states to be able to say 'I can be in any one of these states'. Q', the states of M' will be 2^Q, where a set of states $\{q_1, \ldots q_n\}$ means that if I were playing the game on the original non-deterministic automaton, I could have chosen a path which leads to any of the states q_1 to q_n.

The initial state q_0' is easy to construct: I can only start in state q_0, represented in the new automaton by $\{q_0\}$.

What about the final states F'? In non-deterministic M, we accept a string s, if there exists at least one path from the initial state to a final state. Thus, given a state $q' \in Q'$, we will be able to stop if there is a final state of Q in q' (remember that q' is a state of M' and is thus a subset of Q): $F' \stackrel{\text{df}}{=} \{q' \in Q' \mid q' \cap F \neq \emptyset\}$.

Finally, we come to the transition relation T'. If we lie in a state $q' = \{q_1, \ldots q_n\}$ in Q', it means that we may be in any state q_i. Following a symbol a we will be able to go to any new state q_i' such that $(q_i, a, q_i') \in T$. We will thus have all transitions of the form $(q', a, \{q_2 \mid \exists q_1 \in q' \cdot (q_1, a, q_2) \in T\})$:

$$T' \stackrel{\text{df}}{=} \{(q', a, \{q_2 \mid \exists q_1 \in q' \cdot (q_1, a, q_2) \in T\}) \mid q' \in Q',\ a \in \Sigma\}$$

Note that this language is deterministic, since for every q', a pair, we generate exactly one successor.

□

Since the proof is constructive (we have not only shown that a deterministic automaton exists, but we have actually given instructions on how to construct it), we can now test membership of a string in the language accepted by a finite state automaton by first constructing an equivalent deterministic automaton and then applying the algorithm.

Note that, if n is the number of states in an automaton M, the number of states after making M deterministic is 2^n (the size of the power set of the original states). A 10 state automaton would end up with more than 1000 states while a 20 state automaton would end up with over a 1,000,000 states! Furthermore, it has been shown that we cannot avoid this exponential explosion. It is thus not very efficient to determinise automata, since the size of the resources required increases dramatically.

But how does this apply to regular languages? The following theorem states that regular grammars and finite state machines are equally expressive. In other words, for every regular grammar there is a finite state automaton which recognises the same language and vice versa. The proof is also constructive, allowing us to construct an equivalent finite state automaton from any regular grammar, and then apply the algorithm we have written to determine whether a given string lies in the language generated by the grammar or not.

Lemma: For every finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, there exists a regular grammar $G = \langle \Sigma, N, I, P \rangle$, such that $\mathcal{L}(M) = \mathcal{L}(G)$

Construction: The idea is to use the states of the automaton to remember the current non-terminal in the derivation string. The set of non-terminal symbols will just be the states of the automaton: $N \stackrel{\text{df}}{=} Q$. The start symbol, is now just the initial state: $I \stackrel{\text{df}}{=} q_0$.

The production rules are the most complex part. For every transition in the automaton (q, a, q'), we will introduce the production rule $q \to aq'$. What about termination? For every state in $q \in F$, we should add a transition to stop the derivation $q \to \varepsilon$:

$$P \stackrel{\text{df}}{=} \{q \to aq' \mid (q, a, q') \in T\} \cup \{q \to \varepsilon \mid q \in F\}$$

□

Lemma: For every regular grammar $G = \langle \Sigma, N, I, P \rangle$, there exists a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$ such that $\mathcal{L}(G) = \mathcal{L}(M)$.

Construction: This is more complicated that the opposite conversion. Clearly, we can use the same trick of taking the non-terminal symbols to be the states of automaton M, but we will add a new state τ, the use of which will be explained in a minute: $Q \stackrel{\text{df}}{=} N \cup \{\tau\}$. The initial state is just the start symbol $q_0 \stackrel{\text{df}}{=} I$.

Now consider production rules of the form $A \to aB$. These are easy to translate into (A, a, B). What about termination transitions of the form $A \to a$? State τ now comes into play. We will take τ to be a final state, and add the transition (A, a, τ). Finally, this leaves just epsilon transitions: $A \to \varepsilon$. In this case, we just add A to the set of final states.

$$\begin{aligned} F &\stackrel{\text{df}}{=} \{\tau\} \cup \{q \in N \mid q \to \varepsilon \in P\} \\ T &\stackrel{\text{df}}{=} \{(q, a, \tau) \mid q \to a \in P\} \cup \{(q, a, q') \mid q \to aq' \in P\} \end{aligned}$$

□

Theorem: Finite state automata and regular grammars are equally expressive.

Proof: This follows immediately from the previous two lemmata.

□

2.3 Closure

An important mathematical notion is that of *closure.* A set S is said to be *closed under a mathematical operation op,* if by applying *op* to any element of S always yields an element of S: $\forall x \in S \cdot op(x) \in S$.

Obviously this can be generalised for operators taking more than one parameter in the obvious way. For example, in the binary operator case, we say that S is closed under $\odot$ if $\forall x, y \in S \cdot x \odot y \in S$.

What is so important about closure, you may ask. First of all, we might want closure to ensure that the operator is well defined. Secondly note that S would usually be a set we would like to study in detail. Be it the set of natural numbers, the primes, planar graphs or regular languages, we have usually proved various properties of S. By proving the closure of S under a set of operators, we have an 'invariance' property: no matter how many times we apply the operators, we always end up with an object in S. Thus the result will always satisfy the properties we have proved of the elements of S.

Regular languages have various applications, including parsing, formal verification and hardware design to mention but a few. A number of operators recur in these and other applications of regular languages: taking the union of two languages, the catenation of two languages, iterating a language and, for certain applications, taking the intersection of two languages and the complement of a language. We will prove that the class of regular languages is well behaved under these applications, in the sense that if we apply the operators to regular languages, we always end up with a result which is itself a regular language. Furthermore, we give constructive proofs, which tell us exactly how to calculate the resulting language. This means that, for instance, when we use regular languages to describe temporal properties of systems, the union of two properties is itself a temporal property. Furthermore, thanks to the constructive proof, if we have an algorithm which allows us to verify a property specified as a regular language, we automatically know how to verify a property corresponding to the union of two such others using the same algorithm.

2.3.1 Language Union

Theorem: Regular languages are closed under language union.

Construction: Let L_1 and L_2 be regular languages. We would like to show that $L_1 \cup L_2$ is also regular. Since L_1 is a regular language, there exists a regular grammar $G_1 = \langle \Sigma_1, N_1, I_1, T_1 \rangle$ which produces L_1. Similarly, let $G_2 = \langle \Sigma_2, N_2, I_2, T_2 \rangle$ be a regular grammar producing L_2. We would now like to construct a regular grammar $G = \langle \Sigma, N, I, T \rangle$ such that $L_1 \cup L_2 = \mathcal{L}(G)$.

The alphabet Σ is easy to construct. It is simply the union of the two base alphabets: $\Sigma_1 \cup \Sigma_2$. We now assume that N_1 and N_2 are disjoint. If not, we can always rename the non-terminal symbols of one of them to make sure that they are disjoint.

The trick is now to introduce a new start symbol I which can emulate either I_1 or I_2. This can be easily done using an extended regular grammar.

$$\begin{array}{rcl} \Sigma & \stackrel{\text{df}}{=} & \Sigma_1 \cup \Sigma_2 \\ N & \stackrel{\text{df}}{=} & N_1 \cup N_2 \cup \{I\} \\ P & \stackrel{\text{df}}{=} & P_1 \cup P_2 \cup \{I \to I_1,\ I \to I_2\} \end{array}$$

Since we have previously shown that for every extended regular grammar, there exists a regular grammar with the same language, this construction is sufficient.

To prove that the new grammar produces exactly the union of the original two grammars, we would have two cases (i) take a string in the language of the new grammar. The derivation of the string must start with one of the two new production rules, and then follow rules only from the production rules of one grammar (this would be proved by induction on the length of the derivation) and is thus in the language of one of the grammars; (ii) without loss of generality, take a string in L_1. This means that there exists a derivation of the string in the first grammar. But we know that $I \Rrightarrow I_1$ and hence there exists a derivation of the string in the new grammar.

□

One of the important things to note is that the size of the resulting grammar is just the sum of the sizes of the grammars we started off from (plus a constant), which means that the algorithm is linear, and thus not expensive.

2.3.2 Language Complement

Theorem: Regular languages are closed under language complement.

Construction: Let L be a regular language. We would like to show that the language complement of L^c ($\Sigma^* \setminus L$) is also regular. Since L is a regular language, there exists a regular grammar which produces L. But, for every regular grammar, there exists a finite state automaton which produces the same language. Furthermore, we can also construct a deterministic automaton recognising the same language. Let $M = \langle \Sigma, Q, q_0, F, T \rangle$ be such an automaton.

Note that since M is deterministic, for every state and input pair (q, a), there is at most one q' such that $(q, a, q') \in T$. We will start by augmenting M to make it *total* — for every state and input pair (q, a), there will always exist exactly one q' such that $(q, a, q') \in T$. The trick is to add a dummy state Δ and a transition (q, a, Δ) if there was no q' such that $(q, a, q') \in T$. Furthermore, once we fall inside Δ, we can never escape, guaranteed by adding (Δ, a, Δ) for every a to T. Let $M' = \langle \Sigma, Q', q_0, F, T' \rangle$ be the resulting automaton.

Note that M' is still deterministic, and accepts the same language as M. If we run the membership algorithm we gave on M', we can never fail to match a state and input in the transition relation. In other words, for any string

s, we can follow it all the way through the automaton without getting stuck. Furthermore, $\mathcal{L}(M) = \mathcal{L}(M')$.

Now consider the automaton M^c, exactly like M', except that we now take $Q' \setminus F$ to be the set of final states. If a string s is accepted by M^c, it means that s ends up in a final state of M^c, which is not a final state of M'. Hence M' would not accept s. Conversely, if s is not accepted by M^c, s would lead us to a non-final state of M^c (since M^c, like M' is total), which was a final state of M'. Hence, M^c accepts the complement of M' which, in turn accepts L.

□

What about the complexity? It is sufficient to not that we have to produce a deterministic finite state automaton. Recall that determinising a finite state automaton can be exponential. Hence, this algorithm is not very useful in practice. In fact, it is known that complementation of regular languages is a hard problem, and that we cannot do any better than this (modulo a constant factor, of course).

2.3.3 Language Intersection

Theorem: Regular languages are closed under language intersection.

Proof: Given two regular languages L_1 and L_2, we know that $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. But we know that the complement of of regular language is itself regular, as is the union of two regular languages. Therefore, $(L_1^c \cup L_2^c)^c$ is a regular language, implying that $L_1 \cap L_2$ is regular.

□

Since we use language complementation, the size of the resulting automaton is at least exponential with respect to the size of the original (double exponential, in fact).

2.3.4 Language Catenation

Theorem: Regular languages are closed under language catenation.

Construction: Let L_1 and L_2 be regular languages. We would like to show that $L_1 L_2$ is also regular. Since L_1 is a regular language, there exists a regular grammar $G_1 = \langle \Sigma_1, N_1, I_1, T_1 \rangle$ which produces L_1. Similarly, let $G_2 = \langle \Sigma_2, N_2, I_2, T_2 \rangle$ be a regular grammar producing L_2. We would now like to construct a regular grammar $G = \langle \Sigma, N, I, T \rangle$ such that $L_1 L_2 = \mathcal{L}(G)$. This is what we require to be able to conclude that the catenation of the two regular languages is itself a regular language.

As before, let us assume that N_1 and N_2 are disjoint. What we want to do is to start G_2 whenever G_1 'terminates'. When does G_1 'terminate'? It can do so with a production rule of the form $A \to a$ or one of the form $A \to \varepsilon$. In the first case, we just replace $A \to a$ with $A \to aI_2$. For the second case, we use an extended regular grammar rule: $A \to I_2$.

$$
\begin{array}{lcl}
\Sigma & \stackrel{\text{df}}{=} & \Sigma_1 \cup \Sigma_2 \\
I & \stackrel{\text{df}}{=} & I_1 \\
N & \stackrel{\text{df}}{=} & N_1 \cup N_2 \\
P & \stackrel{\text{df}}{=} & P_2 \cup \{A \to I_2 \mid A \to \varepsilon \in P_1\} \cup \{A \to aI_2 \mid A \to a \in P_1\} \cup \{A \to aB \mid A \to aB \in P_1\}
\end{array}
$$

□

Note that the number of resulting non-terminals is just the sum of the non-terminals in the original languages. Similarly, the number of resulting production rules is the sum of the production rules in the original two languages.

2.3.5 Language Iteration

Theorem: Regular languages are closed under language iteration (L^+ and L^*).

Construction: Let L be a regular language. By definition, there exists a regular grammar G such that $\mathcal{L}(G) = L$. For L^+, we use a trick similar to the one we used in language catenation — for every 'termination' rule, we add rules to 'restart' the grammar:

$$
\begin{array}{lcl}
\Sigma' & \stackrel{\text{df}}{=} & \Sigma \\
I' & \stackrel{\text{df}}{=} & I \\
N' & \stackrel{\text{df}}{=} & N \\
P' & \stackrel{\text{df}}{=} & P \cup \{A \to I \mid A \to \varepsilon \in P\} \cup \{A \to aI \mid A \to a \in P\}
\end{array}
$$

To construct a grammar recognising L^*, we use the law: $L^* = L^+ \cup \{\varepsilon\}$. Let G' be the grammar recognising L^+. We can define G'' recognising L^* as follows:

$$
\begin{array}{lcl}
\Sigma'' & \stackrel{\text{df}}{=} & \Sigma' \\
N'' & \stackrel{\text{df}}{=} & N' \cup \{I''\} \\
P'' & \stackrel{\text{df}}{=} & P \cup \{I'' \to \varepsilon,\ I'' \to I'\}
\end{array}
$$

□

To recognise L^+ we are, at most, doubling the number of production rules, with another extra non-terminal, and two production rules to recognise L^*.

2.4 To Be Or Not To Be a Regular Language

If we start off with regular languages all the standard operators produce just other regular languages! So, this leads to a natural question: are there languages which are not regular? To answer this question we need to look closer

at regular languages and their recognisers. Where does the memory of a finite state automaton lie? It can be found in the state we are currently in. There are no other memory locations the automaton may use to store information. Thus, given an automaton with n states, the automaton can only differentiate between n different situations. Are there any languages which require us to differentiate between an unbounded number of situations? Such languages would be impossible to represent as finite state automata.

2.4.1 The Pumping Lemma

Consider a finite state automaton M accepting a language L. Consider the acceptance path of s in M, s being a *very* long string in L. Since s is very long, we will pass through many states, and since the states are finite we will eventually have to repeat the same state:

$$q_0 \overbrace{\overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_m}{\Rightarrow}}^{s_1} q \overbrace{\overset{a_{m+1}}{\Rightarrow} \ldots \overset{a_l}{\Rightarrow}}^{s_2} q \overbrace{\overset{a_{l+1}}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow}}^{s_3}}^{s} q_n$$

Thus, s could be split into three parts $s = s_1 s_2 s_3$. s_1 can take us from q_0 to a state q, s_2 can take us from q back to itself, and finally s_3 can take us from q to a final state q_n. We can thus repeat s_2 as many times as we want in the middle, and $s_1 s_2^{300} s_3$, $s_1 s_2^7 s_3$ and $s_1 s_2^{438} s_3$, must all be in L.

This means that the finite number of states in an automaton implies a certain repetition (regularity) in the languages we can produce. We will then use this result to prove that certain not-so-repetitive languages cannot be regular.

Lemma: Given an n state finite state automaton M, then for any string $s \in \mathcal{L}(M)$ such that $\sharp(s) \geq n$, there exist s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, $\sharp(s_1 s_2) \leq n$, $\sharp(s_2) \geq 1$ and $\forall k \in \mathbb{N} \cdot s_1 s_2^k s_3 \in \mathcal{L}(M)$.

Proof: Let $s \in \mathcal{L}(M)$, such that $\sharp(s) \geq n$. Since M recognises L, there is a path in M, starting from q_0 and ending in a final state taking as many steps as there are symbols in s to accept:

$$q_0 \overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_m}{\Rightarrow}}^{s} q_m$$

Note that in this path, there are $\sharp(s) + 1$ states. But $\sharp(s) \geq n$ means that we have at least $n+1$ states. But using the pigeon-hole principle, we have n distinct states, and a chain of at least $n+1$ states, means that at the very latest in the first $n+1$ steps, a state must be repeated:

$$q_0 \overbrace{\overset{a_1}{\Rightarrow} q_2 \ldots \overset{a_k}{\Rightarrow}}^{s_1} q \overbrace{\overset{a_{k+1}}{\Rightarrow} \ldots \overset{a_l}{\Rightarrow}}^{s_2} q \overbrace{\overset{a_{l+1}}{\Rightarrow} \ldots \overset{a_m}{\Rightarrow}}^{s_3} q_m$$

This means that we can divide s into three parts s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, where $\sharp(s_1 s_2) \leq n$ (we repeat no longer than after the first $n+1$ steps) and $\sharp(s_2) \geq 1$ (because the state is repeated with at least one symbol in between). Furthermore, $q_0 \overset{s_1}{\Rightarrow} q$, $q \overset{s_2}{\Rightarrow} q$ and $q \overset{s_3}{\Rightarrow} q_m$ where $q_m \in F$.

We now need to prove that $\forall i \in \mathbb{N} \cdot s_1 s_2^i s_3 \in L$. We will start by proving by induction that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i}{\Rightarrow} q$.

Base case i=0: We want to prove that $q_0 \overset{s_1 s_2^0}{\Rightarrow} q$. But $q_0 \overset{s_1}{\Rightarrow} q$, and $s_1 s_2^0 = s_1$, and thus $q_0 \overset{s_1 s_2^0}{\Rightarrow} q$.

Inductive case: Assuming that $q_0 \overset{s_1 s_2^i}{\Rightarrow} q$, we want to prove that $q_0 \overset{s_1 s_2^{i+1}}{\Rightarrow} q$. Since $q_0 \overset{s_1 s_2^i}{\Rightarrow} q$ (inductive hypothesis) and $q \overset{s_2}{\Rightarrow} q$ (see previous diagram), it follows from exercise 1 that $q_0 \overset{s_1 s_2^{i+1}}{\Rightarrow} q$.

Hence, by induction, it follows that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i}{\Rightarrow} q$. Furthermore, since $q \overset{s_3}{\Rightarrow} q_m$, we can conclude that $\forall i \in \mathbb{N} \cdot q_0 \overset{s_1 s_2^i s_3}{\Rightarrow} q_m$, and since $q_m \in F$, it follows that $\forall i \in \mathbb{N} \cdot s_1 s_2^i s_3 \in \mathcal{L}(M)$.

□

Theorem: *(The pumping lemma for regular languages)* For every regular language L, there exists a constant p, called the *pumping length* such that if $s \in L$ and $\sharp(s) \geq p$, then there exist s_1, s_2, s_3 such that $s = s_1 s_2 s_3$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

Proof: Since L is a regular language, there exists finite state automaton M such that $\mathcal{L}(M) = L$. We take p to be the number of states in M, and the rest follows from the previous lemma.

□

2.4.2 Applications of the Pumping Lemma

This pumping lemma may seem to be out of scope. Here, we are trying to show that there are languages which are not regular, and we have started by proving a property which all regular languages must satisfy. How can we proceed to show the non-regularity of a given language? If we can show that a language does not satisfy the pumping lemma property, we know that it cannot be regular.

Example: Prove that $L \overset{\text{df}}{=} \{a^n b^n \mid n \in \mathbb{N}\}$ is not a regular language.

Let us assume that L is regular. Therefore, it must satisfy the pumping lemma for regular languages. Hence, for some p, we know that any string longer than p will start to loop internally.

Consider $s = a^p b^p \in L$. Clearly, $\sharp(s) \geq p$. Now, by the pumping lemma we know that there exist s_1, s_2 and s_3 such that $s_1 s_2 s_3 = a^p b^p$, $\sharp(s_1 s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1 s_2^n s_3 \in L$.

Since $\sharp(s_1 s_2) \leq p$, we know that $s_1 s_2$ is just a repetition of as. From this, together with $\sharp(s_2) \geq 1$, it follows that s_2 is a non-empty repetition of as: $s_2 = a^i$ where $i \neq 0$. Also, $s_1 = a^j$ and $s_3 = a^{p-i-j} b^p$.

But by the pumping lemma, $s_1 s_2^2 s_3 \in L$, where $s_1 s_2^2 s_3 = a^j a^{2i} a^{p-i-j} b^p =$

$a^{p+i}b^p$. Since $i \neq 0$, we know that it is not in language L. Hence, we have reached a contradiction, and thus L cannot be regular.

□

Example: Prove that $L \stackrel{\text{df}}{=} \{ww \mid w \in \{a, b\}^*\}$ is not a regular language.

Assume that L is a regular language. We can apply the pumping lemma, which says that any string longer than some constant p will loop.

Consider string $s = a^pba^pb$. Since $s \in L$ and $\sharp(s) \geq p$, we can apply the pumping lemma. Therefore there must exist a way of splitting s into three parts s_1, s_2 and s_3 such that $s_1s_2s_3 = a^pba^pb$, $\sharp(s_1s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1s_2^ns_3 \in L$.

As in the previous example, we can show that $s_1 = a^i$, $s_2 = a^j$ $(j \geq 1)$ and $s_3 = a^{p-i-j}ba^pb$. But the pumping lemma says that $s_1s_2^2s_3 \in L$, and $s_1s_2^2s_3 = a^ia^{2j}a^{p-i-j}ba^pb$ which can be simplified to $a^{p+j}ba^jb$ which is not in L since $j \geq 1$. Since this leads to a contradiction, L cannot be regular.

□

Example: Prove that $P = \{a^m \mid m \in \mathit{Primes}\}$ is not regular.

As before, assume that L is regular. Let p be the pumping length of L.

Let q be a prime number greater than $p + 1$. Now consider $a^q \in L$. Applying the pumping lemma, we know that there must exist a way of splitting a^q into three parts s_1, s_2 and s_3 such that $s_1s_2s_3 = a^q$, $\sharp(s_1s_2) \leq p$, $\sharp(s_2) \geq 1$ and $\forall n \in \mathbb{N} \cdot s_1s_2^ns_3 \in L$.

Therefore, we can take $s_1 = a^i$, $s_2 = a^j$, $s_3 = a^k$, with $i + j + k = q$ and $j \geq 1$. Furthermore, since $\sharp(s_1s_2) \leq p$ and $\sharp(s) > p + 1$, we know that $k > 1$.

Now consider $s' = s_1s_2^{i+k}s_3$, which the pumping lemma says should be in L. $s' = a^{i+j(i+k)+k} = a^{(i+k)(1+j)}$ But $i + k \neq 1$ (because $k > 1$), and $1 + j > 1$ (since $j > 0$). Hence, $s' = a^l$ such that l can be factorised into $(i+k)$ and $(1+j)$ and is thus not prime. $s' \notin L$ contradicts the pumping lemma, implying that L is not regular.

□

2.4.3 Play the Game

Note that all the proofs take the same form, and are played like a game. We start by assuming that we have a regular language.

The pumping lemma says that a pumping length p must exist. Since I have no control over the length, we assume that it is given to us by the 'other player'.

It is now our turn to construct a string s of length not shorter than p.

Again, it is the opponent's turn to split the string into three parts with certain restrictions on the length of the pieces.

Finally, it is up to us to choose a power to raise the second part of the string such that no matter how the opponent chopped the string, the result cannot be in the original language.

Also note that the pumping lemma can only be used to show that a language is not regular. All it says is that regular languages have a certain property. If I were to tell you that 'All mice are intelligent', and provide you with an example of something intelligent, you cannot conclude it is a mouse (it *may* be a mouse, but not necessarily — Einstein was intelligent, but was not a mouse). On the other hand, if I show you a stupid cow, the fact that it is not intelligent allows you to conclude that it is not a mouse. The pumping lemma follows this reasoning analogously.

2.5 Computability

For which interesting questions about regular languages can we write a program which gives us the answer automatically? We have already written a program which can tell us whether a given string is a member of a regular language (represented as an finite state automaton or regular grammar). We will now look at other questions about regular languages we might want answered.

2.5.1 Is the Language Empty?

Theorem: Given a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, $\mathcal{L}(M) = \emptyset$ if and only if $\mathcal{L}(M)$ contains no string shorter than n, the number of states in Q.

Proof: Clearly, if $\mathcal{L}(M) = \emptyset$, then $\mathcal{L}(M)$ contains no string shorter than n. It is thus sufficient to prove that if $\mathcal{L}(M)$ contains no string shorter than n, then $\mathcal{L}(M)$ is empty.

The proof will follow by contradiction. Assume that $\mathcal{L}(M) \neq \emptyset$, and let $s \in \mathcal{L}(M)$ be a shortest string in the set. Since $\mathcal{L}(M)$ contains no string shorter than n, then $\sharp(s) \geq n$.

Consider the derivation of $s = a_1 a_2 \dots a_m$: $\underbrace{q_0 \overset{a_1}{\Rightarrow} q_1 \overset{a_2}{\Rightarrow} \dots \overset{a_n}{\Rightarrow} q_n}_{n+1 \text{ states}} \overset{a_{n+1}}{\Rightarrow} \dots q_m$

Since the number of distinct states is n, using the pigeon-hole principle, we know that some state must repeat. Therefore, we can divide s into three parts $s = s_1 s_2 s_3$ ($s_2 \neq \varepsilon$) such that for some $q \in Q$ and $q_m \in F$:

$$q_0 \overset{s_1}{\Rightarrow} q \overset{s_2}{\Rightarrow} q \overset{s_3}{\Rightarrow} q_m$$

But this means that $q_0 \overset{s_1 s_3}{\Rightarrow} q_m$, and thus $s_1 s_2 \in \mathcal{L}(M)$. Furthermore, since $s_2 \neq \varepsilon$, $\sharp(s_1 s_2) < \sharp(s_1 s_2 s_3)$, which contradicts our assumption that s was the shortest string in $\mathcal{L}(M)$. Hence, our assumption that $\sharp(s) \geq n$ must be false, implying that there must be an s such that $\sharp(s) < n$.

Therefore, $\mathcal{L}(M) \neq \emptyset$ implies that there exists a string $s \in \mathcal{L}(M)$ such that $\sharp(s) < n$, or equivalently, if $\mathcal{L}(M)$ contains no string shorter than n, then $\mathcal{L}(M)$ is empty.

□

Since the alphabet is finite, we can enumerate the set of strings of length 0 to $n-1$, and for each string we can check whether or not the string lies in the language. Hence, regular language emptiness is computable.

2.5.2 Is the Language Infinite?

Theorem: Given a finite state automaton $M = \langle \Sigma, Q, q_0, F, T \rangle$, $\mathcal{L}(M)$ is finite if and only if the $\mathcal{L}(M)$ contains no string s such that $n \leq \sharp(s) < 2n$ where n is the number of states in Q.

Proof: The proof is split into two parts: (i) if $\mathcal{L}(M)$ contains a string of length at least n, then $\mathcal{L}(M)$ must be infinite; (ii) if L is infinite, then there must exist a string s such that $n \leq \sharp(s) < 2n$. Clearly the 'only if' direction follows from the contrapositive of (i), while the 'if' direction is just the contrapositive of (ii).

Proof of (i): Using the pumping lemma, we have a regular language $\mathcal{L}(M)$, and a string s such that $\sharp(s) \geq n$. By the pumping lemma, we can split s into three $s = s_1 s_2 s_3$ such that for all i, $s_1 s_2^i s_3$ is in $\mathcal{L}(M)$. Since (also by the pumping lemma) $s_2 \neq \varepsilon$, all these strings are distinct, and hence this guarantees that $\mathcal{L}(M)$ is infinite.

Proof of (ii): Assume $\mathcal{L}(M)$ is infinite. The proof will proceed by contradiction, and we will thus start by assuming that there is no string s such that $n \leq \sharp(s) < 2n$. Since the language is infinite, there must be a string of length at least n. Call s one such shortest string. From our assumption, we know that $\sharp(s) \geq 2n$.

By the pumping lemma, $s = s_1 s_2 s_3$ such that $\sharp(s2) > 0$, $\sharp(s_1 s_2) < n$ and for all i, $s_1 s_2^i s_3 \in \mathcal{L}(M)$. Consider $s_1 s_2^0 s_3$ which must thus be in $\mathcal{L}(M)$. From the facts that $\sharp(s) \geq 2n$ and $\sharp(s_1 s_2) < n$, we know that $\sharp(s_3) \geq n$ and thus so is $\sharp(s_1 s_2^0 s_3) \geq n$. But string $s_1 s_3$ is (a) shorter than s (since $\sharp(s_2) \neq 0$), (b) longer than n. This contradicts that s is a shortest string of length at least n.

Therefore, there must exist a string s such that $n \leq \sharp(s) < 2n$.

□

As before, we can enumerate all strings of length at least n, but less that $2n$, and for each one, check whether or not the string lies in the language. This gives us an algorithm to check whether a regular language is infinite or not.

2.5.3 Are Two Regular Languages Equal?

Finally, we treat the question of checking whether two regular languages are equal. Clearly, if we can answer the question whether $L_1 \subseteq L_2$, we can answer the equality question.

But in set theory, we know that $L_1 \subseteq L_2 \Leftrightarrow L_2^c \cap L_1 = \emptyset$. But we know how to construct the regular grammar/automaton describing the complement and intersection of two languages. Furthermore, we have just shown how to check whether a language described by a regular grammar or automaton can be checked for emptiness. We can thus check the equality of two regular languages.

2.6 Exercises

1. *(Moderate)* Prove that if $q_1 \stackrel{s_1}{\Rightarrow} q_2$ and $q_2 \stackrel{s_2}{\Rightarrow} q_3$, then $q_1 \stackrel{s_1 s_2}{\Rightarrow} q_3$.

2. *(Moderate)* Prove that, given a regular grammar G, all strings reachable in G are of the form Σ^* or $\Sigma^* N$.

3. *(Easy)* Doubling a string s replaces each symbol a in s by aa:

$$\begin{array}{rcl} 2(\varepsilon) & \stackrel{\text{df}}{=} & \varepsilon \\ 2(as) & \stackrel{\text{df}}{=} & aa\, 2(s) \end{array}$$

   As usual, doubling a language can be defined in terms of this operator $2(L) \stackrel{\text{df}}{=} \{2(s) \mid s \in L\}$.

   Give a construction to show that regular languages are closed under doubling.

4. *(Moderate)* Give a construction to show that regular languages are closed under language reversal.

5. *(Easy)* Prove that regular languages are closed under language (set) difference. Discuss the complexity of the constructed result.

6. *(Difficult)* Given a string $s \in \Sigma^*$, and symbol $a \in \Sigma$, we define *the hiding of a in s* as follows:

$$\begin{array}{rcl} \varepsilon \dagger a & \stackrel{\text{df}}{=} & \varepsilon \\ (bs) \dagger a & \stackrel{\text{df}}{=} & \begin{cases} s \dagger a & \text{if } b = a \\ b(s \dagger a) & \text{otherwise} \end{cases} \end{array}$$

   This can be generalised to work over whole languages: $L \dagger a \stackrel{\text{df}}{=} \{s \dagger a \mid s \in L\}$.

   Give a construction to show that regular languages are closed under hiding.

7. Give an algorithm to calculate $\overline{\varepsilon}(A)$ on a regular grammar. What is the space and time complexity of the algorithm?

8. *(Easy-Moderate)* Prove that the following languages are not regular:

   - $\{w \mid w \in \{a, b\}^*,\ w = w^R\}$
   - $\{a^{n^2} \mid n \in \mathbb{N}\}$
   - $\{1^n + 1^m = 1^{n+m} \mid n, m \in \mathbb{N}\}$

9. *(Difficult)* The linear repetition of a string s repeats the symbols of s in an increasing fashion:

$$\nearrow (a_1 a_2 \ldots a_n) \stackrel{\text{df}}{=} a_1^1 a_2^2 \ldots a_n^n$$

Similarly, the linear repetition of a language L is defined in terms of this string operator:

$$\nearrow (L) \stackrel{\text{df}}{=} \{\nearrow (s) \mid s \in L\}$$

Prove that regular languages are not closed under linear repetition.

Hint: Use can use the language $L = \{(ab)^n \mid n \in \mathbb{N}\}$ and show that $(\nearrow (L))^R$ is not regular.

10. *(Easy)* Write the algorithms to check for language emptiness and language finitude, based on the solutions given in this chapter.

11. *(Easy)* Prove that all finite languages are regular.

12. *(Moderate)* An engineer decides to use a finite state automaton to generate test inputs for his program. Each symbol represents a keypress from the user. The program requests three numbers (written in base 10) from the user (each followed by the pressing of the return key).

    (a) Give a finite state automaton which generates all numbers even numbers.

    (b) Construct another automaton to generate all numbers divisible by 3. (**Hint:** In base 10, a number is divisible by 3 if and only if the sum of its digits is itself divisible by 3).

    (c) Hence or otherwise, discuss how the engineer can generate an automaton to recognise all numbers divisible by 6.

    (d) How would she then combine all these to construct the test pattern automaton which recognises a sequence of three numbers: the first being divisible by 2, the second by 3, and the third by 6?