Contents

Chapter 1

Prior Knowledge

From Foundations of Computer Science

1.4 Languages

An *alphabet* Σ is a finite set of symbols, e.g., $\Sigma = \{a, b\}$, $\Sigma = \{a, b, c\}$ or $\Sigma = \{0, 1\}$. The set of all finite strings over an alphabet Σ is denoted by Σ^*. For example,

$$\{a, b\}^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, \ldots\}$$

Here, the symbol Λ denotes the empty string, i.e., the string consisting of zero symbols. Note that, although the elements of Σ^* have finite length, there are infinitely many of them. A *language* L over an alphabet Σ is a subset of Σ^*. Nothing more, nothing less. For example, Σ^* itself is a language over Σ, and so is the empty set $\emptyset$. Also, the set $\{\Lambda\}$ is a language (over any alphabet Σ).

Note the difference between the empty string Λ, the empty set $\emptyset$, and the set $\{\Lambda\}$ consisting of the empty string only. The empty string Λ is a string, whereas $\emptyset$ and $\{\Lambda\}$ are sets of strings, and thus languages. The set $\emptyset$ contains zero strings, whereas $\{\Lambda\}$ contains one string, the empty string.

The canonical order is an ordering of strings, where shorter strings come before longer strings, and where strings of the same length are 'alphabetically' ordered. For example,

$$\Lambda, a, b, aa, ab, ba, bb, aaa, \ldots$$

is a list of the ('first') elements of Σ^* in canonical order. The canonical order is defined for any language. For example, the (first) elements of $Pal = \{x \in$

$\{a,b\}^* \mid x = x^r\}$ (the set of palindromes over $\{a,b\}$) in canonical order are:

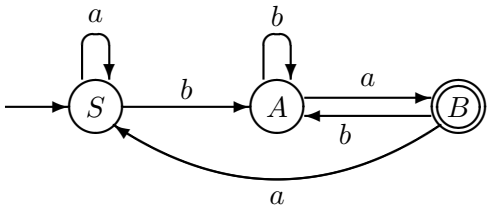
$\Lambda, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, \ldots$

2.1 Finite Automata

The simplest model of computation we studied is the finite automaton (FA). It has a finite number of states. These states are the only type of memory the automaton has. That is, the only way for the automaton to remember anything relevant about the input read so far, is by the state it is in.

Example.

An FA accepting $\{a,b\}^*\{ba\}$



This finite automaton accepts all strings in $\{a,b\}^*$ that end with ba. The tree states keep track of how much of the desired suffix ba we have read so far. In state S, we have not read anything of this suffix. In state A, we have just read b. In state B, we have just read ba. That is, the string we have read so far, ends with ba, and thus we can accept this string. That is why B is the (only) accepting state. ■

2.2.3 Combining languages

Finite automata accepting particular languages may be combined to accept more complex languages:

Theorem 2.7 *(Product construction for finite automata)*
Let $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$ *and* $M_2 = (Q_2, \Sigma, q_2, A_2, \delta_2)$ *be finite automata over the (same) alphabet* Σ*, accepting languages* L_1 *and* L_2*, respectively. Let* M *be the finite automaton* $(Q, \Sigma, q_0, A, \delta)$*, where*
– $Q = Q_1 \times Q_2$
– $q_0 = (q_1, q_2)$
– $\delta(\ (p,q), \sigma) = (\ \delta_1(p,\sigma), \delta_2(q,\sigma)\ )$ *for all* $(p,q) \in Q_1 \times Q_2$.

1. *If* $A = \{(p,q) \mid p \in A_1$ *or* $q \in A_2\}$*, then* $L(M) = L(M_1) \cup L(M_2)$

2. *If* $A = \{(p,q) \mid p \in A_1$ *and* $q \in A_2\}$*, then* $L(M) = L(M_1) \cap L(M_2)$

3. *If* $A = \{(p,q) \mid p \in A_1$ *and* $q \notin A_2\}$*, then* $L(M) = L(M_1) - L(M_2)$

`[M]` *Th. 2.15*

Hence, the class of regular languages is closed under union, intersecton and difference. The construction of M from M_1 and M_2, taking the product of the state sets Q_1 and Q_2, initial state (q_1, q_2) and the combination of the two transition functions δ_1 and δ_2, is called the product construction. Before we (partially) proof Theorem 2.7, we consider an example.

Example 2.8 Consider the two finite automata M_1 and M_2 in Figure 2.1. It is easy to see that M_1 accepts in state e all strings over $\{a, b\}$ with an even number of a's, and that M_2 accepts in state b all strings ending with b. When we apply the product construction with $A = \{eb\}$, we obtain finite automaton M, accepting all strings with an even number of a's and ending with b, the intersection of the two languages. ■

Proof of Theorem 2.7. We give a detailed proof for claim 1. The proofs for claims 2 and 3 are analogous.

One can prove by induction (see Exercise 2.11) that for every $x \in \Sigma^*$ and every $(p,q) \in Q$,

$$\delta^*((p,q), x) = (\delta_1^*(p,x), \delta_2^*(q,x)) \tag{2.1}$$

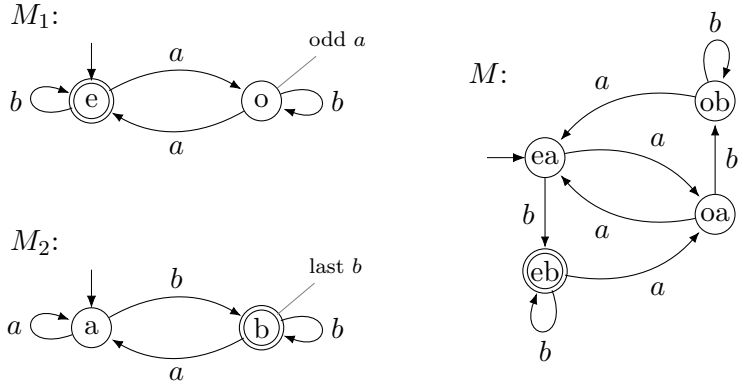


Figure 2.1: Product construction for two simple finite automata M_1 and M_2, yielding a finite automaton M with $L(M) = L(M_1) \cap L(M_2)$. See Example 2.8.

Assume that $A = \{(p, q) \mid p \in A_1 \text{ or } q \in A_2\}$, which is the case in claim 1.

We first prove that $L(M_1) \cup L(M_2) \subseteq L(M)$. Consider an arbitrary string $x \in L(M_1) \cup L(M_2)$. Then either $x \in L(M_1)$, or $x \in L(M_2)$ (or both). Without loss of generality, assume that $x \in L(M_1)$. Hence, $\delta_1^*(q_1, x) \in A_1$.

By (2.1),

$$\delta^*(q_0, x) = \delta^*((q_1, q_2), x) = (\delta_1^*(q_1, x), \delta_2^*(q_2, x)) \tag{2.2}$$

This state is in A, because the first component is in A_1. Hence, $x \in L(M)$.

We now prove that $L(M) \subseteq L(M_1) \cup L(M_2)$. Consider an arbitrary string $x \in L(M)$. Again, equation (2.2) is valid. Because $x \in L(M)$, the resulting, combined state must be in A. By definition of A, we must have either $\delta_1^*(q_1, x) \in A_1$, or $\delta_2^*(q_2, x) \in A_2$ (or both). Without loss of generality, assume that $\delta_1^*(q_1, x) \in A_1$. Then apparently, $x \in L(M_1)$, and thus also $x \in L(M_1) \cup L(M_2)$.

We have found that both $L(M_1) \cup L(M_2) \subseteq L(M)$ and $L(M) \subseteq L(M_1) \cup L(M_2)$. We can thus conclude that $L(M) = L(M_1) \cup L(M_2)$. □

The product construction always works, but it does not necessarily yield a minimal finite automaton for the language involved.

Example 2.9 Let us reconsider Example 2.8. The finite automaton M resulting from the construction is shown again in 2.2. States ob and oa both correspond to strings containing an odd number of a's. For ob, the strings

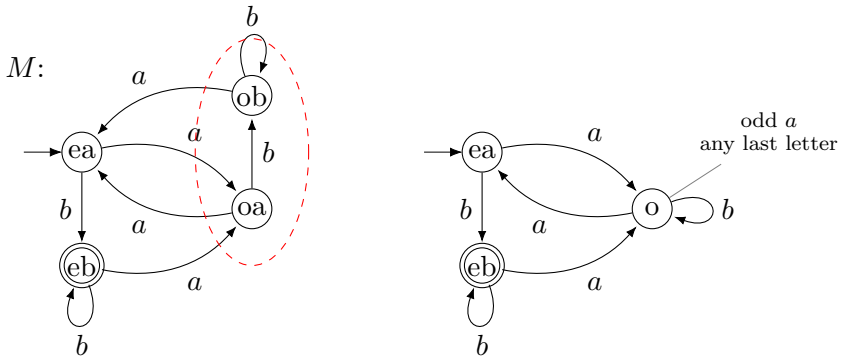


Figure 2.2: The finite automaton M from Figure 2.1 and an equivalent finite automaton with fewer states. States ob and oa have been merged. See Example 2.9.

end with b; for oa, they end with a. If we wish to accept strings with an even number of a's ending with b, then the difference between ob and oa is irrelevant, because we need to read another a anyway. The states can be combined into a single state o, corresponding to strings with an odd number of a's, ending with any letter.

Note that if the set of accepting states of M had been {eb, ea, ob}, corresponding to $L(M_1) \cup L(M_1)$, then we could not merge states oa and ob, because the latter would be an accepting state, where the former would not. Instead, we could merge states ea and eb in that case. ■

Example 2.10 Consider the 3-state finite automata M_1 and M_2 in Figure 2.3. It is easy to see that M_1 accepts all strings over $\{a, b\}$ not containing a substring aa, and that M_2 all strings over $\{a, b\}$ ending with ab. The construction yields a finite automaton M with $3 \cdot 3 = 9$ states. However, three states are unreachable, and three other states can be merged into a single garbage state. [M] E 2.16 ■

Example 2.11 Consider the finite automata M_1 and M_2 in Figure 2.4. M_1 accepts all strings over $\{a, b\}$ containing (at least) a substring ab, M_2 accepts all strings over $\{a, b\}$ containing (at least) a substring bba. When we apply the product construction to the 3-state FA M_1 and the 4-state FA M_2, we obtain a 12-state FA, where three of the twelve states turn out to be

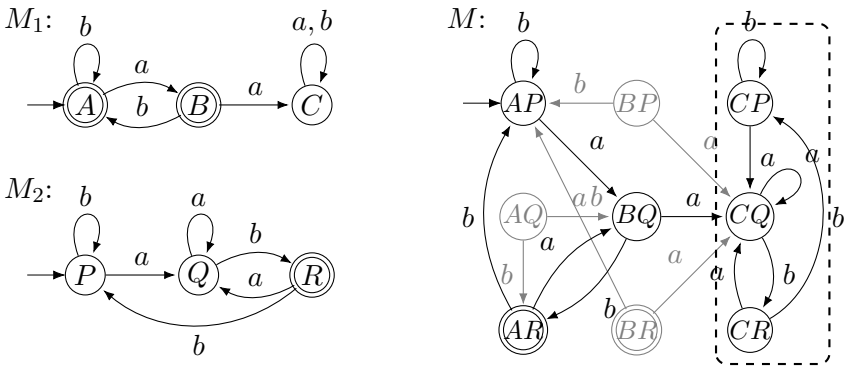


Figure 2.3: Product construction for two simple finite automata M_1 and M_2, yielding a finite automaton M with $L(M) = L(M_1) \cap L(M_2)$. States BP, AQ and BR are unreachable, and states CP, CQ and CR might be merged. See Example 2.10.

unreachable. When we omit these three states, we obtain the automaton M that is also shown in Figure 2.4. The five accepting states can be merged, because if a string contains either of the two substrings ab and bba, then it does not matter anymore whether or not it also contains the other. [M] E. 2.18 ■

Example 2.12 Consider the two finite automata M_1 and M_2 in Figure 2.5. M_1 accepts all strings over $\{a, b\}$ that start and end with an a. M_2 accepts all strings over $\{a, b\}$ of even length. Application of the production construction to the 4-state FA M_1 and the 2-state FA M_2 yields an 8-state FA, where state Λo is unreachable (obviously!). When we leave out this state, we obtain the automaton M that is also shown in Figure 2.5. Here, the two states corresponding to strings starting with a b can me merged into a single garbage state. Also the states corresponding to strings of odd length which start with a can be merged, because we need another letter to make the length even anyway. ■

2.4 The Pumping Lemma

Regular languages are the languages that can be accepted by a finite automaton. The pumping lemma for regular languages intuitively states that

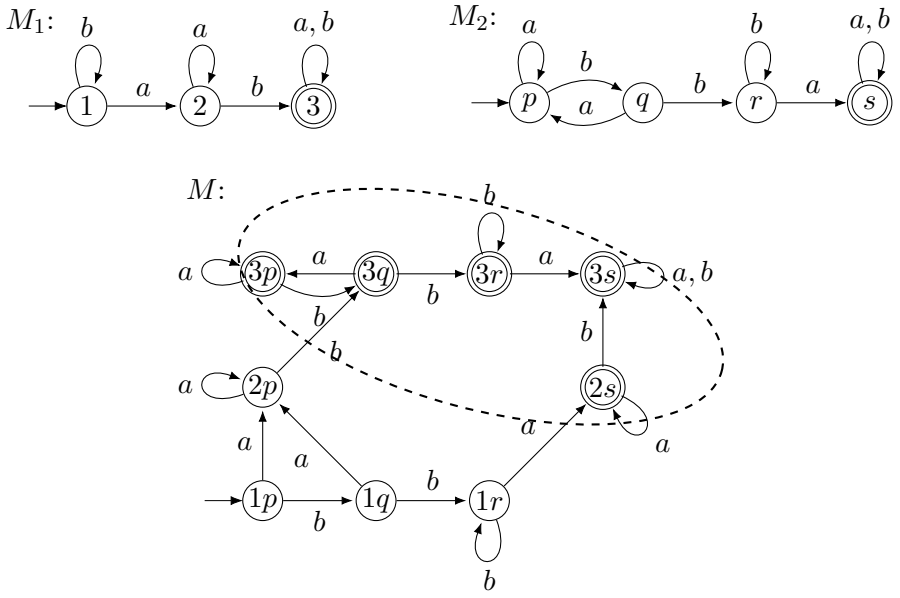


Figure 2.4: Product construction for two simple finite automata M_1 and M_2, yielding a finite automaton M with $L(M) = L(M_1) \cup L(M_2)$ (with unreachable states left out). States $3p$, $3q$, $3r$, $3s$ and $2s$ might be merged. See Example 2.11.

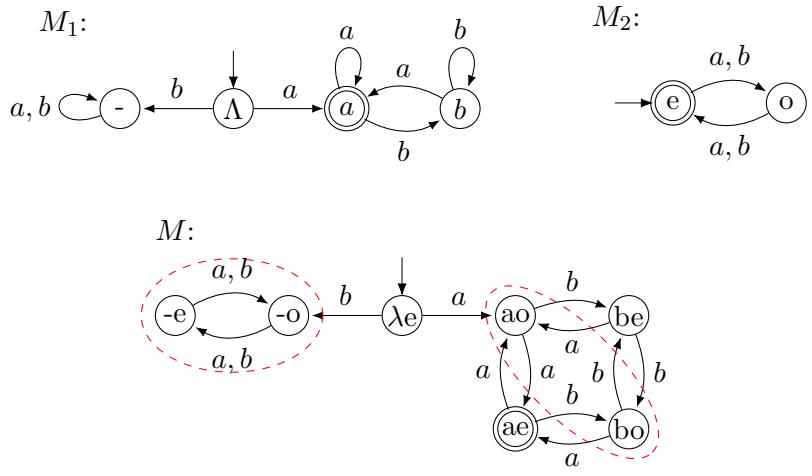


Figure 2.5: Product construction for two simple finite automata M_1 and M_2, yielding a finite automaton M with $L(M) = L(M_1) \cap L(M_2)$ (with an unreachable state left out). States -e and -o might be merged, and states ao and bo might also be merged. See Example 2.12.

any string in a regular language which is long enough, can be 'pumped up and down'. In more formal terms:

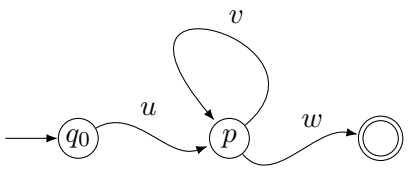
Theorem 2.13 *(Pumping lemma for regular languages)*
Suppose L is a language over the alphabet Σ. If L is accepted by a finite automaton M and if $n \geq 1$ is the number of states of M, then for every string $x \in L$ satisfying $|x| \geq n$ there exist strings $u, v, w \in \Sigma^*$, such that $x = uvw$ (i.e., x can be split into u, v and w) and the following three conditions are true:

1. *$|uv| \leq n$ (in particular, substring v occurs within the first n letters of x)*

2. *$|v| \geq 1$ (i.e., substring v is not empty)*

3. *for $m = 0, 1, 2, \ldots$, the string $uv^m w$ belongs to L (i.e., we can pump up (or down) v)*

Hence we can pump string x within its first n letters.
Proof. The proof of this pumping lemma is very intuitive:



[M] Fig. 2.28

Figure 2.6: Intuitive proof of the pumping lemma.

Assume that indeed, language $L \subseteq \Sigma^*$ is accepted by a finite automaton M with $n \geq 1$ states. Let x be an arbitrary element of L such that $|x| \geq n$. Since x is accepted by M, x follows a path through M ending in an accepting state. M has only n different states, whereas x consists of at least n letters. Hence, the first n steps of the path (actually, any n steps of the path) through M for x must visit at least one state multiple times. Let p be such a state, and let v be the substring of x traversed between the first and the second visit of state p. Let u and w be the substrings of x, before and after v, respectively. Indeed $x = uvw$, and $|uv| \leq n$, see Figure 2.6.

Then instead of the path through M for $x = u \cdot v \cdot w$, we could as well traverse the path for $u \cdot w$ (skipping the subpath from state p back to itself), for $u \cdot v \cdot v \cdot w$ (traversing this subpath two times), for $u \cdot v \cdot v \cdot v \cdot w$ (traversing this subpath three times), etcetera. Each of these alternative paths ends in the same state as the path for x, which is an accepting state since $x \in L$. We conclude that all strings $uv^m w$ are accepted by M and thus are elements of L. □

Most often, the pumping lemma for regular languages is used to prove that some language L is *not* regular, simply because it does *not* satisfy the pumping lemma. To this end, one shows that pumping (up and/or down) a particular string $x \in L$ yields a string that is *not* an element of L, while it should be according to the pumping lemma. We first give an example.

Example 2.14 We prove that the language $L = \{a^i b^i \mid i \geq 0\}$ is not regular, by contradiction.

Assume that $L = \{a^i b^i \mid i \geq 0\}$ *is* regular. Let M be a finite automaton such that $L = L(M)$, and let n be the number of states of M.

Take $x = a^n b^n$. Then obviously, $x \in L$, and $|x| = 2n \geq n$. According to the pumping lemma, there should exist a decomposition $x = uvw$ such that $|uv| \leq n$ with $|v| \geq 1$, and $uv^m w \in L$ for every m.

Consider such a decomposition $uvw = x = a^n b^n$. Because $|uv| \leq n$, the prefix uv consists of a's only. Hence, also v consists of a's only, at least one, since $|v| \geq 1$. In particular $v = a^k$ for some k with $1 \leq k \leq n$.

Now, take $m = 0$, i.e., we pump down the string x by deleting the substring v. This will delete $k \geq 1$ a's from x. So uv^0w is of the form $a^{n'}b^n$ with $n' < n$. This string is not in L, which contradicts the pumping lemma. Hence, our assumption that L is regular must be false. In other words, L is not regular. (Note that $m \geq 2$ would also yield a contradiction.)
[M] Exmp. 2.30 ■

In general, a proof that a language $L \subseteq \Sigma^*$ is not regular, using a contradiction with the pumping lemma for regular languages, looks as follows:

1. Assume that L *is* regular, and thus that L is accepted bij a finite automaton M. Let n be the number of states of n.

2. *Choose* a proper string $x \in L$, such that $|x| \geq n$.

   Note that we do not know what exactly the number of states n is. It might be 17 or 1,000, or any positive integer number. To make sure that $|x| \geq n$, the string x must be defined in terms of n.

   According to the pumping lemma, all strings $x \in L$ satisfying $|x| \geq n$ can be pumped. In particular, the string x that we have chosen. If we can prove that this particular string *cannot* be pumped, then we have a contradiction.

   There may be strings $x \in L$ with $|x| \geq n$ that can indeed be pumped. Finding at least one x that cannot be pumped, is enough to have a contradiction. Therefore, we have to choose a *proper* string $x \in L$ with $|x| \geq n$.

3. Consider an *arbitrary* decomposition $x = uvw$ of x into substrings u, v and w, such that (1) $|uv| \leq n$ and (2) $|v| \geq 1$.

   According to the pumping lemma, *there exists* such a decomposition that also has a third property: (3) for all $m \geq 0$, the string uv^mw belongs to L. The pumping lemma does not say what exactly the strings u, v and w are. We cannot assume anything about them, except that $|uv| \leq n$ and $|v| \geq 1$.

   We cannot simply *choose* particular strings u, v and w for this. For a contradiction with the pumping lemma, we have to prove that there does not exist *any* decomposition that also has property (3). Finding a contradiction with property (3) for one particular decomposition does

not necessarily mean that every decomposition leads to a contradiction.

4. Now given this arbitary decomposition $x = uvw$, *choose* a proper value m, and show that the string $uv^m w$ does *not* belong to L. That would contradict the pumping lemma, because according to the pumping lemma, all strings $uv^m w$ should belong to L.

   There may be values m for which $uv^m w$ does belong to L. As long as we find one value for which this is not the case, we have a contradiction. That is why we have to choose a *proper* value m.

   For some languages, we may have to consider different cases, for different possibilities for the decomposition. For example, if the decomposition looks like this, then this value m yields a contradiction, and if the decomposition looks like that, then that (possibly different) value m yields a contradiction. If for each possible decomposition we can find a proper value m, then we have a contradiction with the pumping lemma.

5. Conclude that L does not satisfy the pumping lemma for regular languages, and thus cannot be regular.

This five-step process can also be derived from an alternative formulation of the pumping lemma, using quantifiers $\exists$ and $\forall$:

Theorem 2.15 *(Pumping lemma for regular languages, alternative)*
If L *is a regular language,* then
$\exists$ *there exists a constant* $n \geq 1$ *(namely, the number of states of a finite automaton accepting* L*)*
such that
$\forall$ *for every* $x \in L$
with $|x| \geq n$
$\exists$ *there exists a decomposition* $x = uvw$
with (1) $|uv| \leq n$,
and (2) $|v| \geq 1$
such that
$\forall$ *(3) for all* $m \geq 0$, $uv^m w \in L$

This formulation can be read like: *If* p, *then* q, for predicates p and q. We know from the introductory course on logics, that such a statement is equivalent to: *If* not q, *then* not p. When we apply this to Theorem 2.15, we obtain

Theorem 2.16 If
∀ *for every* $n \geq 1$
∃ *there exists* $x \in L$
with $|x| \geq n$
such that
∀ *for every decomposition* $x = uvw$
with (1) $|uv| \leq n$,
and (2) $|v| \geq 1$
∃ *(3) there exists* $m \geq 0$,
such that
$uv^m w \notin L$
then L *is not a regular language.*

In fact, our five-step process follows exactly this formulation.

The choice of a string x

The first step in our five-step process that requires creativity, is choosing a string x (step 2). In general, you may use two rules of thumb for this, which we will illustrate for the language $L = \{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$:

1. Choose a string x with a simple structure. If x has a simple structure, then there are probably not too many possibilities for the decomposition that need to be considered, perhaps only one. For example, $a^{n+1}b^n$ has a simpler structure than $(ab)^n a = abab \ldots aba$.

   In fact, the string $x = (ab)^n a$ *does* satisfy the pumping lemma. That is, there does exist a decomposition uvw of x, for which v can safely be pumped. Take, e.g., $u = \Lambda$, $v = ab$ and $w = (ab)^{n-1}a$. Then, indeed $x = uvw$, $|uv| \leq n$ (assuming $n \geq 2$), and $|v| \geq 1$. Moreover, for any $m \geq 0$, $uv^m w = \Lambda(ab)^m(ab)^{n-1}a \in L$.

   Of course, if we would *choose* $u = \Lambda$, $v = a$ and $w = b(ab)^{n-1}a$, then again $x = uvw$, $|uv| \leq n$ and $|v| \geq 1$. This time, if we pump v up or down, we obtain a string $uv^m w$ that is not in L. For example, if we take $m = 0$ (we pump down the string), we obtain $\Lambda\Lambda b(ab)^{n-1}a$, which is not in L, because the number of a's in the string equals the number of b's.

   The fact that we have a particular decomposition for which we cannot pump x does not mean that x contradicts the pumping lemma. Only if *each* decomposition uvw of x (with $|uv| \leq n$ and $|v| \geq 1$) has this property, then we have such contradiction. Therefore, we have

to consider every possible decomposition to conclude that we have a contradiction.

Because the earlier decomposition with $u = \Lambda$, $v = ab$ and $w = (ab)^{n-1}a$ can safely be pumped, $x = (ab)^n a$ is not a proper choice to find a contradiction.

2. Choose a string $x \in L$ that precisely satisfies the requirements of the language, i.e., which is 'close to falling out of the language'. For example, $a^{n+1}b^n$ is precisely in our example language L. Decreasing the number of a's by 1, or increasing the number of b's bij 1, is enough to obtain a string outside L. In contrast, $a^{2n}b^n$ is also in L, but is much further away from falling out of the language.

   In fact, the string $x = a^{2n}b^n$ is not suitable to find a contradiction with the pumping lemma, because there does exist a decomposition uvw of x for which we can pump v in all possible ways and stay in the language. We can, e.g., take $u = \Lambda$, $v = a$ and $w = a^{2n-1}b^n$, for which indeed $x = uvw$, $|uv| \leq n$ and $|v| \geq 1$. For each $m \geq 0$, the string $uv^m w = a^m a^{2n-1}b^n$ is in L (assuming $n \geq 2$, which is necessary if $m = 0$).

   We could even take $u = \Lambda$, $v = a^{n-1}$ (assuming $n \geq 2$) and $w = a^{n+1}b^n$. Also in that case, $uv^m w \in L$ for all $m \geq 0$.

   Only the decomposition with $u = \Lambda$, $v = a^n$ and $w = a^n b^n$ yields a string that is not in L. If, in this case, we pump the string down, i.e., we consider $m = 0$, then the string $uv^m w = a^n b^n$, which is not in L.

   Again, the existence of a decomposition uvw for which we cannot pump v in all possible ways, does not mean that we have a contradiction with the pumping lemma. We must have this property for every possible decomposition. Therefore, $a^{2n}b^n$ is not suitable for finding a contradiction.

Example 2.17 The language $AeqB = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$ is not regular.
This can also be proved by contradiction with the pumping lemma, just like we did for $L = \{a^i b^i \mid i \geq 0\}$ in Example 2.14. In fact, we can choose the exact same string x and use the exact same arguments why this string cannot be pumped.

Alternatively, we may use a closure property of the regular languages from Section 2.2 for this proof. Assume that *AEqB is* regular. Then consider the language $L_2 = \{a\}^*\{b\}^*$. This language can easily be accepted by a finite

automaton, and thus is regular. Then by Theorem 2.7, also the intersection of $AEqB$ and L_2 must be regular. However, this intersection is $L = \{a^i b^i \mid i \geq 0\}$, which we have seen is not regular. We thus have a contradiction, and our assumption that $AEqB$ is regular must be false. ■

Example.
The language $SimplePal = \{xcx^r \mid x \in \{a, b\}^*\}$ is not regular, because pumping the string $x = a^n c a^n$ (again, with n the size of the finite automaton from the pumping lemma) would yield a string with fewer (if pumped down) or more (if pumped up) than n a's to the left of the c, but still n a's to the right of the c. ■

3.1 Regular Languages and Regular Expressions

A regular language over an alphabet Σ is a language derived from the empty language $\emptyset$ and the languages $\{a\}$ (with $a \in \Sigma$), by means of the operators union, concatenation and Kleene star.

A regular expression over an alphabet Σ is an expression over the empty expression and expressions a (with $a \in \Sigma$), by means of the operators $+$, concatenation and Kleene star. In principle, the order of application of the operators in a regular expression is determined by brackets. Brackets may, however, be left out when they are redundant with respect to the following rules of precedence: Kleene star comes before concatenation, which in turn comes before $+$.

Each regular language can be described by a regular expression. Usually, there are more than one regular expressions describing the same regular language. Conversely, each regular expression describes a unique regular language.

Example.
An example of a regular language is $\{a, b\}^*\{ba\}$. A regular expression describing this language is $(a + b)^*ba$. Another regular expression describing the same language is $(b + a)^*ba$. ■

3.2 Nondeterministic Finite Automata

By default, a finite automaton is deterministic, which means that for each state p and each symbol a from the input alphabet Σ, there is one transition from p with label a (to some state q). There do not exist Λ-transitions. In other, more formal terms, we have a transition *function* $\delta : Q \times \Sigma \to Q$, where Q is the set of states of the finite automaton.

A *nondeterministic finite automaton* (NFA) is an extension of a finite automaton. For each state p and each symbol a from the input alphabet Σ, it has zero or more transitions from p with label a (to different states q). Each state p may also have zero or more Λ-transitions (to different states q). In other, more formal terms, we have a transition function $\delta : Q \times (\Sigma \cup \{\Lambda\}) \to 2^Q$, where 2^Q is the set of all subsets (including $\emptyset$) of Q.

3.3 The Nondeterminism in an NFA Can Be Eliminated

As said, a *nondeterministic finite automaton* (NFA) is an extension of a finite automaton. Each FA is (in fact) also an NFA. Hence, each regular language, i.e., each language that is accepted by an FA, can also be accepted by an NFA. It is less obvious that each language that is accepted by an NFA can also be accepted by an FA. One might imagine that NFAs are 'stronger' than FAs, i.e. that NFAs can accept more languages than just the regular languages. This is, however, not the case.

Using the so-called subset construction, each NFA M can be transformed into an FA M' accepting the same language. The term 'subset construction' refers to the states in the FA M' that is constructed: the states of M' correspond to subsets of states of M, namely all states that M may be in after reading a certain string.

3.4 Kleene's Theorem, Part 1

In the foregoing, we already mixed up the terms 'language accepted by a finite automaton' and 'regular language'. It is not obvious that each regular language as defined in Section 3.1 can be accepted by a finite automaton and vice versa. Kleene's theorem states that this is the case, indeed.

Part 1 of the theorem states that each regular language L over an alphabet Σ can be accepted by a finite automaton. The proof is by induction on the structure of the regular expression describing L. First, we give NFAs (in fact, FAs) accepting the basic regular languages $\emptyset$ and $\{a\}$ (with $a \in \Sigma$). Next, given two NFAs M_1 and M_2 accepting regular languages L_1 and L_2, respectively, we describe how to construct an NFA accepting $L_1 \cup L_2$, an NFA accepting $L_1 \cdot L_2$ and an NFA accepting L_1^* (or L_2^*). As each regular language is the result of a finite number of applications of (some of) these operations on the basic regular language, each regular language can be accepted by an NFA. But then each regular language can also be accepted by an FA (see Section 3.3).

3.5 Kleene's Theorem, Part 2

Part 2 of Kleene's theorem states that each language that is accepted by a finite automaton, can be described by a regular expression, and thus is a regular language. Indeed, the terms 'language accepted by a finite automaton'

and 'regular language' are synonyms.

4.2 Context-Free Grammars

A context-free grammar G is a tuple (V, Σ, S, P) where V and Σ are two disjoint sets of symbols, V is the set of variables (or non-terminals), Σ is the set of terminals, $S \in V$ is the start variable and P is the set of productions. Each production is of the form $A \to \beta$, where $A \in V$ and $\beta \in (V \cup \Sigma)^*$.

A production $A \to \beta$ can be applied to a string α, which means that an occurrence of the variable A in α is replaced by β. For example, if $\alpha = \alpha_1 A \alpha_2$, then we may write: $\alpha_1 A \alpha_2 \ \Rightarrow \ \alpha_1 \beta \alpha_2$. The language generated by G is the set of strings $x \in \Sigma^*$ that can be derived from the start variable S by successively applying productions from P. This language is denoted by $L(G)$. A language generated by a context-free grammar is called a context-free language.

Example.
The context-free grammar (V, Σ, S, P), with $V = \{S\}$, $\Sigma = \{a, b, c\}$ and the following productions:

$$S \to aSa \mid bSb \mid c$$

generates the language $\mathit{SimplePal} = \{xcx^r \mid x \in \{a, b\}^*\}$. For example, the string $abbcbba$ is derived as follows:

$$S \ \Rightarrow \ aSa \ \Rightarrow \ abSba \ \Rightarrow \ abbSbba \ \Rightarrow \ abbcbba$$

■

This example illustrates that context-free grammars can generate languages that are non-regular (see Section 2.4).

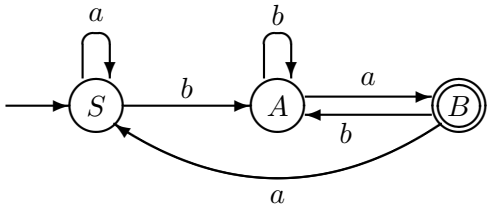
4.3 Regular Languages and Regular Grammars

A regular grammar (V, Σ, S, P) is a context-free grammar whose productions are restricted to two general forms: either $A \to \sigma B$ or $A \to \Lambda$, where $A, B \in V$ and $\sigma \in \Sigma$. For example, a production $S \to aSA$ is not allowed in a regular grammar.

A finite automaton $(Q, \Sigma, q_0, A, \delta)$ accepting a language L can easily be transformed into a regular grammar (V, Σ, S, P) generating the same language. Just take $V = Q$, $S = q_0$,
for each $X, Y \in V$ and $\sigma \in \Sigma$, if $\delta(X, \sigma) = Y$, then add production $X \to \sigma Y$ to P,

and for each $X \in A$ (i.e., each accepting state X), add production $X \to \Lambda$ to P.

Example.
Consider again the following FA accepting $\{a, b\}^*\{ba\}$:



The construction described yields the regular grammar $(\{S, A, B\}, \{a, b\}, S, P)$, with productions

$$S \to aS \mid bA \qquad A \to bA \mid aB \qquad B \to bA \mid aS \mid \Lambda$$

■

With exactly the converse construction, each regular grammar generating a language L, can be transformed into a (nondeterministic!) finite automaton accepting the same language.

This implies that regular grammars can generate precisely the regular languages. Because the class of regular grammars is a subset of the class of context-free grammars, all regular languages can be generated by context-free grammars. However, as we have seen in Section 4.2, context-free grammars can also generate non-regular languages.

4.4 Derivation Trees

A derivation tree represents the structure of a string derived in a context-free grammar. For each production $A \to X_1 \dots X_n$ applied in the derivation, the corresponding node labelled by A has n ordered children labelled by $X_1, \dots, X_n$ respectively, read from left to right. If $n = 0$ (i.e., the production is $A \to \Lambda$), the node labelled by A has a child labelled by Λ.

4.5 Simplified Forms and Normal Forms

As said, the class of regular grammars is a subset of the class of context-free grammars. A subset that can generate only regular languages.

There exist other classes of context-free grammars which can generate (nearly) all context-free languages. One such class consists of the context-free grammars in Chomsky normal form. In such a grammar (V, Σ, S, P), the productions are (again) restricted to two general forms: either $A \to BC$ or $A \to \sigma$, where $A, B, C \in V$ and $\sigma \in \Sigma$. It can be proved that for each context-free grammar G, there exists a context-free grammar G' in Chomsky normal form, such that $L(G') = L(G) \setminus \{\Lambda\}$. That is, apart from the empty string, $L(G)$ can be generated by a context-free grammar in Chomsky normal form.

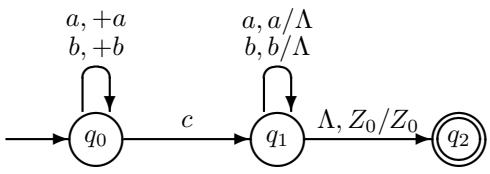
5.1 Definitions and Examples (of Pushdown Automata)

A pushdown automaton (PDA) is a finite automaton, extended with a stack. Initially, the stack contains only the initial stack symbol Z_0. Transitions in the automaton not only depend on the current state and the next input letter to be read, but also on the current top symbol on the stack. As a result of a transition, this top symbol is replaced by a string of symbols. If this string is empty, the top symbol is effectively just popped from the stack.

By default, pushdown automata may have Λ-transitions, and may be nondeterministic: there may be more than one transitions from the same state on the same input letter (or Λ) and the same top stack symbol.

A second source of nondeterminism is the combination of Λ-transitions and 'letter-transitions': for a given state and a given top stack symbol, there may be both Λ-transitions and transitions with an input letter. When, in this case, the next letter to be read is indeed this input letter, we may (or may not) postpone reading this letter by following a Λ-transition.

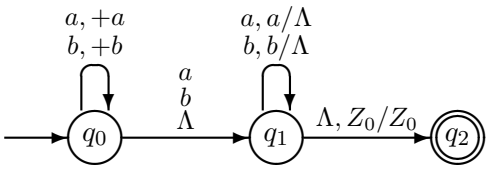
Example 5.3 A Pushdown Automaton Accepting $SimplePal = \{xcx^r \mid x \in \{a, b\}^*\}$.



At the initial state q_0, symbols a and b read from the input are pushed onto the stack. When the special middle symbol c is read, the PDA moves to state q_1 without altering the stack. At state q_1 symbols are popped from the stack, only if they equal the next input symbol. Otherwise, the PDA just crashes. When all symbols a and b have been popped from the stack (the top symbol is Z_0, again), the PDA moves to accepting state q_2 with a Λ-transition. ■

Example 5.7 A Pushdown Automaton Accepting *Pal*

$$Pal = \{x \in \{a, b\}^* \mid x = x^r\}$$



The elements of language *Pal* do not have a special middle symbol c. Hence, there is no way for the PDA to recognize what the middle of the word is. Therefore, while pushing symbols onto the stack in state q_0, it nondeterministically decides what to do with the next input symbol, say a:

- The PDA may assume that a is still part of the first half of the input string, and therefore also push it onto the stack in state q_0.

- The PDA may assume that a is the middle symbol of the input string (which means that the length of the input string is odd), and move to state q_1, while reading a.

- The PDA may assume that a is the first symbol of the second half of the input string (which means that the length of the input string is even), and move to state q_1 without reading a yet (corresponding to the Λ-transition.

The options for input symbol b in state q_0 are the same. Once the PDA has reached state q_1, it continues its operation in exactly the same way as the PDA in Example 5.3. ■

5.2 Deterministic Pushdown Automata

As said before, by default, a pushdown automaton may be nondeterministic. A *deterministic* pushdown automaton is a pushdown automaton in which

- for each combination of state q, stack symbol X and input symbol a, there is at most one transition, and
- for each combination of state q and stack symbol X, if there exists an Λ-transition, then there does not exist any transition for this state q, stack symbol X and any input symbol a.

The PDA from Example 5.3 is deterministic. On the other hand, the PDA from Example 5.7 is not deterministic. There are, e.g., two transitions in state q_0, any stack symbol and input symbol a. Moreover, there is an Λ-transition from q_0 to q_1 for any stack symbol.

There exist languages that can be accepted by a PDA but cannot be accepted by a deterministic PDA. An example of such a language is Pal, that was accepted by the PDA in Example 5.7.

Hence, unlike for finite automata, a deterministic pushdown automaton is not equally powerful as a (general) pushdown automaton.

5.3 A PDA from a Given CFG

Every context-free language can be accepted by a pushdown automaton. We have two constructions from a given CFG G to a PDA accepting $L(G)$:

- a construction yielding the so-called nondeterministic top-down PDA corresponding to G, $NT(G)$.
- a construction yielding the so-called nondeterministic bottom-up PDA corresponding to G, $NB(G)$.

Students following the course Compilerconstructie may recognize $NT(G)$ in the top-down parser and $NB(G)$ in the bottom-up parser occurring in that course.

5.4 A CFG from a Given PDA

Every language that can be accepted by a pushdown automaton can be generated by a context-free grammar. This implies that the context-free languages are exactly the languages that can be accepted by PDAs. Recall,

however, that there exist languages, like Pal, that can be accepted by a PDA, but not by a deterministic PDA. Hence, not all context-free languages can be accepted by deterministic PDAs.

The construction of a CFG from a given PDA M is pretty complicated. First we convert M into a PDA M', such that $L(M) = L_e(M')$. That is, the language that M accepts 'in the normal way', with accepting states, is equal to the language that M' accepts by empty stack.

Now suppose that $M' = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ (where the set A of accepting states is irrelevant, because M' accepts strings by empty stack). Every string $x \in L_e(M')$ has a computation in M' from initial configuration (q_0, x, Z_0) to a configuration (q, Λ, Λ) for some state $q \in Q$. M' starts with a stack of height 1, consisting only of the initial stack symble Z_0. In the course of the computation, the stack may increase and/or decrease. Processing string x eventually causes M' to lower its stack by a single symbol, resulting in an empty stack.

We wish to simulate the computations of M' by derivations in a context-free grammar $G = (V, \Sigma, S, P)$. Such a derivation starts with a string consisting only of the start variable S. In the course of the derivation new variables may appear in the string, even multiple variables at the same time. A string $x \in L(G)$ consists only of terminal symbols. That is, all variables have eventually disappeared.

Hence, if $x \in L_e(M')$, then there exists a computation in M' for x by which we get rid of all stack symbols. On the other hand, if $x \in L(G)$, then there exists a derivation in G by which we get rid of all variables (and are left with terminal string x).

This similarity suggests to use the stack symbols of M' as variables in G. However, this would be too simple, because the replacement of stack symbols by other (strings of) stack symbols in M' is restricted by the states that you visit. After a number of steps in the computation, you may only continue the computation from the state that you have just arrived in.

To model this in our context-free grammar, we extend the variables to triples $[pXq]$, where X is the stack symbol we wish to eliminate, p is the state where we start the process of eliminating X, and q is the state where we arrive at right after eliminating X.

6.5 The Pumping Lemma for Context-Free Languages

Although the class of context-free languages is larger than the class of regular languages, there are still many languages that are not context-free. For example, it is plausible that there cannot be a PDA accepting the language $AnBnCn = \{a^i b^i c^i \mid i \geq 0\}$, and hence no CFG generating the language. A PDA is able to count the a's on its stack, by pushing a symbol onto the stack for every a it reads. However, in order to check that the number of b's following the a's is equal, it has to pop all these symbols. By then, it has no way to check that also the number of c's following the b's is equal.

It is also plausible that there cannot be a PDA accepting the language $XX = \{xx \mid x \in \{a, b\}^*\}$. A PDA might push all symbols of the first half of its input string onto the stack. It may even guess the middle of the string. By then, the only way to compare the first letter of the second half of the string to the first letter of the first half of the string (which resides at the bottom of the stack), is by removing all other symbols from the stack. After that the PDA cannot compare the other symbols of the second half to the other symbols of the first half of the string, anymore.

In Section 2.4, we used the pumping lemma for regular languages to prove that certain languages are not regular. We use the pumping lemma for context-free languages to prove that certain languages are not context-free.

The pumping lemma for regular languages was based on the path through a finite automaton for a long sting x. In contrast, the pumping lemma for context-free languages is not based on the operation of a PDA accepting a language, but on the derivation of a string in a CFG.

If a string derived with the grammar is long enough, there should be a nonterminal A in this derivation that generates itself, together with some non-empty substrings. That is, there should be a derivation of our string, containing the following sentential forms:

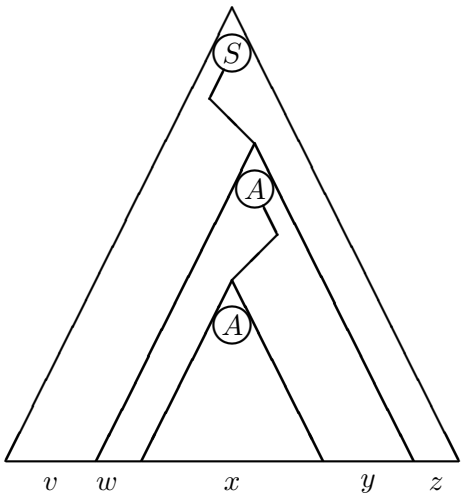
$$S \Rightarrow^* v\underline{A}z \Rightarrow^* v\ w\underline{A}y\ z \Rightarrow^* vw\ x\ yz$$

Apparently the substring wAy can be derived from A. But then this part of the derivation can be repeated, over and over again:

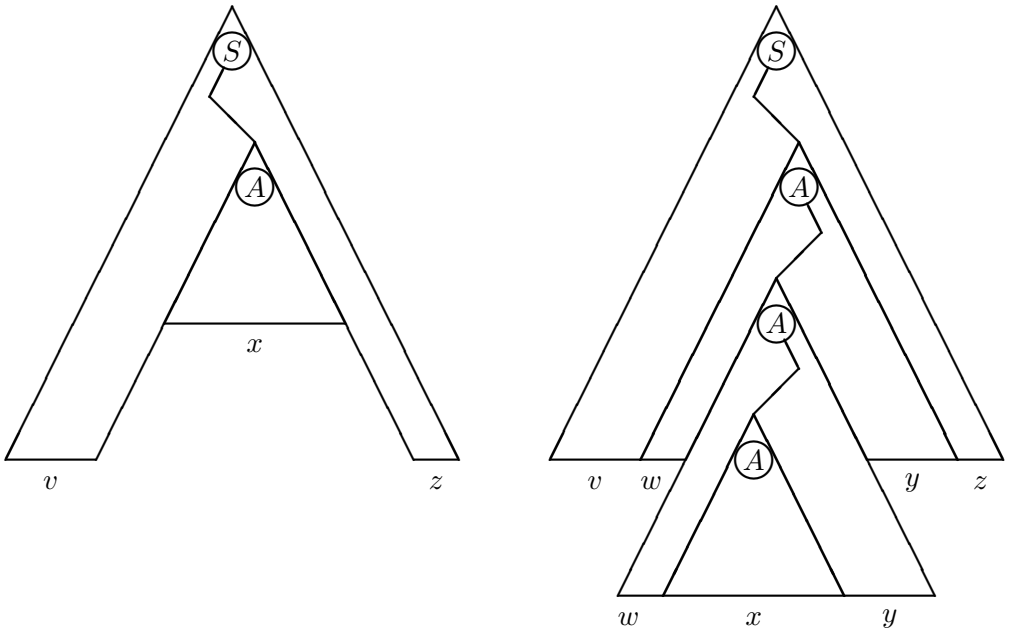
$$S \Rightarrow^* v\underline{A}z \Rightarrow^* v\ w\underline{A}y\ z \Rightarrow^* vw\ wAy\ yz \Rightarrow^* vw^m xy^m z$$

The underlying cause of this is exactly the *context-freeness* of the grammar: no matter what the context is in which the variable A occurs, we can derive the same strings from it. It does not matter if it occurs in vAz or in $vwAyz$.

We can also visualize this argument with derivation tree. If the sting derived is long enough, then there must be a node in the tree labelled by a variable A, which has as a descendent another node labelled by A, as in



But then we could obtain another valid derivation tree by replacing one subtree rooted by A by the other. This may result in the following new derivation trees:



In the left picture, we have replaced the larger subtree by the smaller subtree, yielding a shorter string vxz In the right picture, we have replaced the smaller subtree by the larger subtree, yielding a longer string $vwwxyyz$. This way, we can pump up (or down, as in the first case) a string.

A formal description of this result is:

Theorem 6.1 *The Pumping Lemma for Context-Free Languages*

Suppose L is a context-free language. Then there is an integer n so that for every $u \in L$ with $|u| \geq n$, u can be written as $u = vwxyz$, for some strings v, w, x, y and z satisfying

1. $|wy| > 0$
2. $|wxy| \leq n$
3. *for every* $m \geq 0$, $vw^m xy^m z \in L$

This pumping lemma is used to prove that, a.o., both the language $AnBnCn$ and the language XX are not context-free. There are many more non-context-free languages.