Introduction to Finite-State Techniques

Lecture notes to master class "Finite-State Techniques in NLP",
July 8-12, 1996, Groningen (The Netherlands)

Mark-Jan Nederhof *

University of Groningen
The Netherlands
markjan@let.rug.nl

Abstract

In this document, we describe some fundamentals of formal language theory pertaining to regular languages, finite automata, rational transductions and related notions.

1 Introduction

Formal language theory studies *formal languages*, which are languages which can be described very precisely. For example, programming languages are formal languages, since what is a correct program and what is not is determined very precisely by a compiler for that programming language, which will accept some programs as correct and some others as incorrect (because of typos, or perhaps because of some more serious conflict with the syntax of that programming language). As this example suggests, in formal language theory one is foremost interested in the set of strings that a language contains; one is generally *not* interested in what these strings mean.

Natural languages, e.g. English, Dutch or Japanese, are not formal, because no well-defined boundary exists between correct sentences of English, Dutch or Japanese and those that are incorrect. Yet, useful formal *approximations* can be found for such languages. Such formal definitions approximating phenomena of natural languages can be encoded into computer programs and be used for automated processing of natural language (i.e. computer-aided translation, spell checking, grammar checking, etc.), or these formal descriptions can be used in their own right by linguists to express theories about natural language.

Regular languages are very primitive languages, yet allow approximations of interesting phenomena of natural languages. In this document we will discuss regular languages, and a simple machine model for processing regular languages, i.e. the finite automata. Further, we will describe how we can specify relations between regular languages. First we will fix some terminology about formal languages.

Some of the material in this document was adopted from [LP81] and [Ber79].

*Supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802.

2 Formal languages

By an *alphabet* we mean a finite set of *symbols*. An example of an alphabet is the Roman alphabet $\{a, b, c, \ldots, x, y, z\}$, but other alphabets may be used too. For example, we may use a *binary alphabet* of the form $\{0, 1\}$, containing only two symbols. An alphabet is a set of symbols to which we want to restrict ourselves when we discuss some formal language.

Given an alphabet, we can construct a string of symbols from the alphabet. For example, "*road*" is a *string over* the Roman alphabet, since it consists of 4 symbols selected from the set $\{a, b, c, \ldots, x, y, z\}$. There is one special string that we can make irrespective of the chosen alphabet, and that is the empty string. Because "nothingness" is so hard to print, we introduce the symbol ϵ to indicate the empty string.

We can put two strings next to each other to make a new string. This is called *concatenation*, and we use the operator $\circ$ to denote that two strings should be concatenated. For example, $road \circ runner$ denotes the concatenated string *roadrunner*. If no confusion can result, we omit the $\circ$ operator; so if x and y denote strings, then we may write xy instead of $x \circ y$ to denote the concatenated string.

The *length* of a string w, denoted $|w|$, is exactly what you would expect: the number of symbols occurring in it. We of course count a symbol twice if it occurs in the string twice, e.g. $|runner| = 6$.

We can now formulate a number of properties about strings. In the following, x, y and z denote arbitrary strings.

$$\begin{aligned} x(yz) &= (xy)z \\ x \circ \epsilon &= x \\ \epsilon \circ x &= x \\ |\epsilon| &= 0 \\ |xy| &= |x| + |y| \end{aligned}$$

The first equation states that concatenation is *associative*, which means that if you glue three strings together, then it does not matter in which order you do this. For example, both $(bike \circ repair) \circ shop = bikerepair \circ shop = bikerepairshop$ and $bike \circ (repair \circ shop) = bike \circ repairshop = bikerepairshop$ result in the same string. Note that because of associativity, both $x(yz)$ and $(xy)z$ may be written simply as xyz without causing confusion.

We denote the set of all strings over an alphabet Σ as Σ^*. The asterisk in Σ^* is called the *Kleene star*; we will encounter it once more below.

A *language over* an alphabet is a subset of Σ^*, in other words, a language is a certain selection from the strings that one can make with the alphabet.

As an example, suppose that we take alphabet $\Sigma = \{a, b\}$. The set Σ^* consists of all strings that we can make from a's and b's. One example of a language over Σ is the set L of all strings consisting of equal numbers of a's and b's. For example $aabbba \in L$, since the number of a's equals the number of b's, whereas $abbaa \notin L$, since it has one a too many.

The operation of concatenation can also be applied to languages. Suppose that L_1 and L_2 are languages, then $L_1 \circ L_2$ denotes the set of all strings that we can make by taking a string from L_1 and concatenating it with some string from L_2. Formally

$$L_1 \circ L_2 = \{vw \mid v \in L_1 \wedge w \in L_2\}$$

For example, if

$$L_1 = \{w \in \{a, b\}^* \mid w \text{ has an even number of } a\text{'s}\}$$

$$L_2 = \{w \in \{a,b\}^* \mid w \text{ consists of an } a \text{ followed by zero of more } b\text{'s}\}$$

then

$$L_1 \circ L_2 = \{w \in \{a,b\}^* \mid w \text{ has an odd number of } a\text{'s}\}$$

Similarly, the Kleene star can be applied to languages. If L is a language, then L^* denotes the set of all strings which result from taking zero or more strings from L and concatenating them. Formally:

$$L^* = \{w \mid w = w_1 \circ w_2 \circ \ldots \circ w_k,\ k \geq 0,\ w_1, \ldots, w_k \in L\}$$

A related construction of a language from another language is

$$L^+ = \{w \mid w = w_1 \circ w_2 \circ \ldots \circ w_k,\ k > 0,\ w_1, \ldots, w_k \in L\}$$

We have for any language L:

$$\begin{aligned} L^+ &= L^* \circ L \\ \epsilon &\in L^* \end{aligned}$$

Note that $\emptyset^* = \{\epsilon\}$. Further, $\epsilon \in L^+$ implies $\epsilon \in L$.

The complement of a language, denoted $\overline{L}$, is the set of all strings over the alphabet (it is essential that we know what the alphabet is!) that are not in L. Formally

$$\overline{L} = \{w \in \Sigma^* \mid w \notin L\}$$

3 Regular expressions

The *regular expressions* over an alphabet Σ are defined inductively as follows:

1. $\emptyset$ is a regular expression.
2. Each symbol from Σ is a regular expression.
3. If α and β are regular expressions, then so is $(\alpha \circ \beta)$.
4. If α and β are regular expressions, then so is $(\alpha \cup \beta)$.
5. If α is a regular expression, then so is $(\alpha)^*$.

An example of a regular expression is $(a \cup b)^* ab^*$. Some brackets have been omitted to improve readability; we have also omitted $\circ$, as explained in the previous section. We assign higher precedence to concatenation than to union, so e.g. $ab \cup c$ is interpreted as $((ab) \cup c)$; and we assign higher precedence to Kleene star than to concatenation or union, so e.g. $ab^* \cup c^*$ is interpreted as $((a(b)^*) \cup (c)^*)$.

Symbols $a \in \Sigma$ in regular expressions are interpreted as the singleton languages $\{a\}$. The other operators ($\emptyset$, $*$, $\cup$ and $\circ$) represent the corresponding operations on languages and sets. Thus, the regular expression $(a \cup b)^* ab^*$ denotes all strings over $\{a,b\}$ containing at least one a.

The languages that can be described using regular expressions are called *regular languages*.

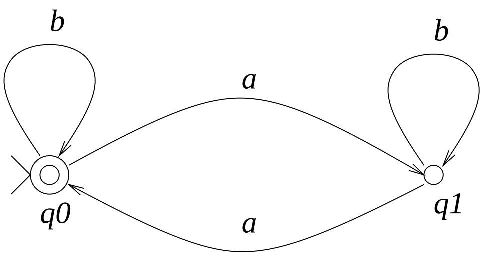

Figure 1: A deterministic finite automaton

4 Finite automata

Regular expressions are a *generative* formalism, i.e. it is easy to build a string in the language described by a regular expression, but it is less easy to decide whether a given string is in the language described by a regular expression.

In this section we give a model for language recognition devices for regular languages, i.e. we give a mathematical model of a device that decides whether a given string is in a fixed regular language.

There are two kinds of finite automata, viz. deterministic and nondeterministic ones. Both read the input string from left to right, each time being in a certain state. The next state after reading a symbol from the input string depends on the previous state and on the symbol read. In a deterministic finite automaton, this next state is uniquely determined.

4.1 Deterministic finite automata

Formally, a *deterministic finite automaton* is a 5-tuple $M = (K, \Sigma, \delta, s, F)$, where K is a finite set of *states*, of which s is the *initial state*, in which the automaton is when it starts to read the input, F is a subset of K representing the *final states*, i.e. those that indicate that the input is correct when it has been completely read. The function δ from $K \times \Sigma$ to K computes a new state from a previous state and an input symbol.

An example is the deterministic finite automaton $M = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_0\})$, where δ is given by

$$\begin{array}{rcl} \delta(q_0, a) & = & q_1 \\ \delta(q_0, b) & = & q_0 \\ \delta(q_1, a) & = & q_0 \\ \delta(q_1, b) & = & q_1 \end{array}$$

More friendly looking than this formal definition of δ is the picture in Figure 1. States are depicted as vertices, and an arrow from state q to state q' labelled a represents that $\delta(q, a) = q'$. The initial state is marked by a ">"-sign, and the final states are marked by an extra circle around the state.

The semantics of a finite automaton is given as follows. We define a *configuration* to be a tuple consisting of a state and part of the input that has not been read yet.[1] We

[1]The formal term for this is *suffix* of the input. In general, a string y is a *suffix* of a longer string xy.

define the binary relation $\vdash$ between configurations as: $(q, aw) \vdash (q', w)$ if and only if $\delta(q, a) = q'$. The relation $\vdash$ denotes that one configuration can result from another by applying one step of the automaton. The result of applying zero or more steps is denoted by $\vdash^*$. Formally, $\vdash^*$ is what we call the transitive and reflexive closure of $\vdash$.

The automaton starts with the configuration (s, v), where v is the input. The input is accepted when $(s, v) \vdash^* (q, \epsilon)$, some final state q.

For the above example, we describe how the automaton accepts input *abaabba* by

$$\begin{array}{rcl} (q_0, abaabba) & \vdash & (q_1, baabba) \\ & \vdash & (q_1, aabba) \\ & \vdash & (q_0, abba) \\ & \vdash & (q_1, bba) \\ & \vdash & (q_1, ba) \\ & \vdash & (q_1, a) \\ & \vdash & (q_0, \epsilon) \end{array}$$

The first configuration in this sequence consists of the initial state and the input. After the automaton has read all input symbols, it is in state q_0. Since this is a final state, this means the input is to be considered correct.

In general, the example automaton above accepts all input strings in $\{a, b\}^*$ that have an even number of a's. The other strings over alphabet $\{a, b\}$ will not be accepted, because the automaton will end in a configuration (q_1, ϵ), and q_1 is not a final state.

Related to $\vdash^*$ is the function $\hat{\delta}$, which results from generalizing δ to apply to *strings* over the alphabet instead of to single symbols. Intuitively, $\hat{\delta}(q, w)$ is the state reached from state q by reading w. Formally:

$$\begin{array}{rcl} \hat{\delta}(q, \epsilon) & = & q \\ \hat{\delta}(q, aw) & = & \hat{\delta}(\delta(q, a), w) \end{array}$$

The set of all strings that a certain automaton accepts is called the language *accepted by* that automaton. Formally, the language accepted by an automaton can be defined by $\{v \in \Sigma^* \mid (s, v) \vdash^* (q, \epsilon),\ \text{some } q \in F\}$, or alternatively by $\{v \in \Sigma^* \mid \hat{\delta}(s, v) \in F\}$.

4.2 Nondeterministic finite automata

For nondeterministic finite automata the new state obtained after reading an input symbol may not be uniquely determined. That is, perhaps we may have $(q, aw) \vdash (q', w)$, but at the same time $(q, aw) \vdash (q'', w)$, for another state q''.

Formally, a nondeterministic finite automaton is a 5-tuple $M = (K, \Sigma, \Delta, s, F)$, where Δ is no longer a function but a relation. This *transition relation* is a finite subset of $K \times \Sigma^* \times K$. Note that *strings* over the alphabet Σ may be involved instead of individual symbols from Σ. This additional extension, next to nondeterminism, is not essential, but is often a handy feature.

One may draw similar figures for nondeterministic finite automata as in the case of Figure 1, with the difference that several arrows may leave a state that are not mutually exclusive, e.g. two arrows that leave a single state may be labelled by the same symbol a, or more generally, one arrow may be labelled by a string which is a prefix of the string with which another arrow is labelled.[2]

[2] We say v is a *prefix* of string w if w can be written as vx, some string x. Cf. "suffix".

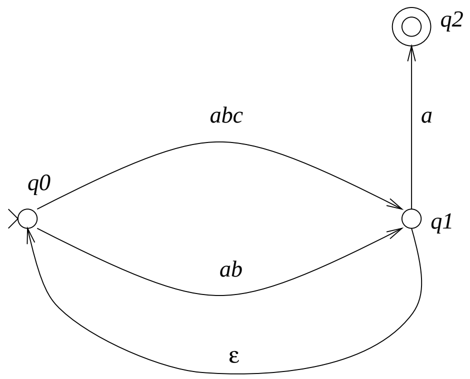

Figure 2: A nondeterministic finite automaton

Further, it may be the case that for some symbols in the input, no next state may be found. In that case the automaton may get stuck before reaching the end of the input.

Figure 2 provides an example. We have $\Delta = \{(q0, abc, q1),\ (q0, ab, q1),\ (q1, \epsilon, q0),\ (q1, a, q2)\}$.

The formal definition for $\vdash$ is now that $(q, vw) \vdash (q', w)$ if there is a tuple $(q, v, q') \in \Delta$. The nondeterminism of finite automata is sometimes called *angelic*, in the sense that if there is one way to obtain some configuration (q, ϵ), $q \in F$, from the initial configuration, then the input is considered correct, irrespective of other derivable configurations (q, ϵ), $q \notin F$, and those where the automaton gets stuck because no transitions are applicable. Formally, a string w is *accepted* by the automaton if $(s, w) \vdash^* (q, \epsilon)$, where s is the initial state, and q is some final state.[3]

4.3 The powerset-construction

Nondeterminism is a useful feature, but has the disadvantage that it cannot be implemented on a computer. If we restrict ourselves to classical mechanics (no quantum mechanics), then nondeterminism does not even exist in the "real" world. The only thing we can hope to do therefore is to build mathematical models for it, where either a machine "guesses right" in the case of nondeterminism (cf. the infamous "little birdie"), or where the world splits itself into multiple worlds. This second option we explore further here.

Consider the nondeterministic finite automaton in Figure 3.

There are a number of ways to process the input *aaa*. These are given in the form of a *search tree* in Figure 4. The labels of the nodes are configurations. The root of the tree is the initial configuration. The tree branches at nondeterminism. The input is accepted if there is at least one leaf labelled with a final configuration, i.e. (q, ϵ) with $q \in F$. In the example, there are three ways to reach a final configuration.

In the search tree, a number of layers can be distinguished. E.g. the top-most layer consists of all configurations that can be reached before any input has been read (in particular, we may apply transitions of the form (q, ϵ, q')). The second layer results from reading the first input symbol (also here, transitions of the form (q, ϵ, q') might have been applied, were they applicable). Because transitions of the form (q, v, q'), where $|v| > 1$,

[3]The opposite of angelic nondeterminism is *demonic* nondeterminism, whereby a process is considered to fail if at least one of the possible ways to perform the process fails.

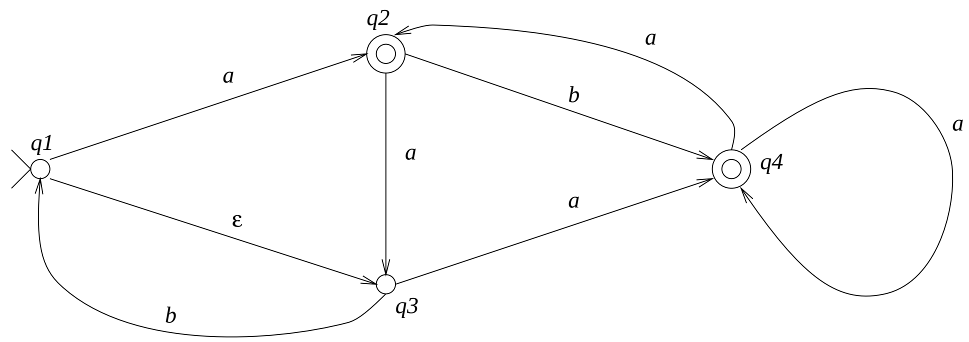

Figure 3: Another nondeterministic finite automaton

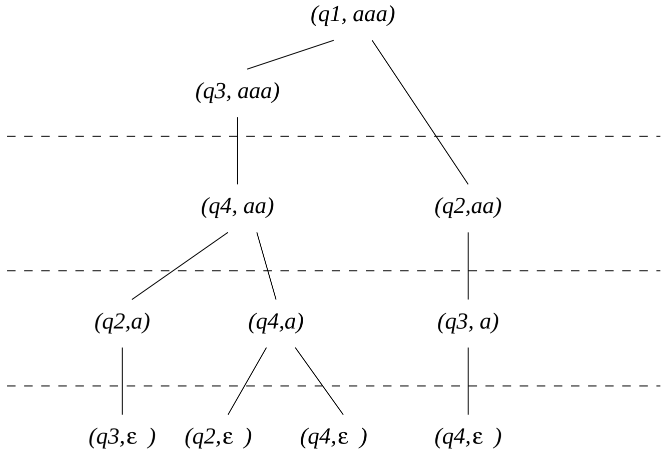

Figure 4: Search tree

form a small snag in this layer structure, we will assume in this section that these do not exist.[4]

The observation about the layers in the search tree gives us the intuition how we can transform a nondeterministic finite automaton into a deterministic one: introduce new states that represent *sets* of states of the original automaton. The initial state of the transformed automaton represents all states of the old automaton that are reachable without reading any input (cf. the top-most layer in Figure 4). In the new automaton there is a transition labelled a from state q to state q' if the states of the old automaton that q' represents can be reached from the states of the old automaton that q represents by reading a.

We call this the powerset-construction, since the states in the new automaton represent subsets of the set of states in the old automaton. Somewhat more formally, given $M = (K, \Sigma, \Delta, s, F)$, where we assume $\Delta \subseteq K \times (\Sigma \cup \{\epsilon\}) \times K$, we construct $M' = (K', \Sigma, \delta, s', F')$, where $K' = 2^K$, i.e. the set of all subsets of K (also called the

[4] Any nondeterministic finite automaton can be transformed into one where such transitions do not occur: any arrow labelled v is replaced by $|v|$ arrows labelled by the consecutive symbols in v.

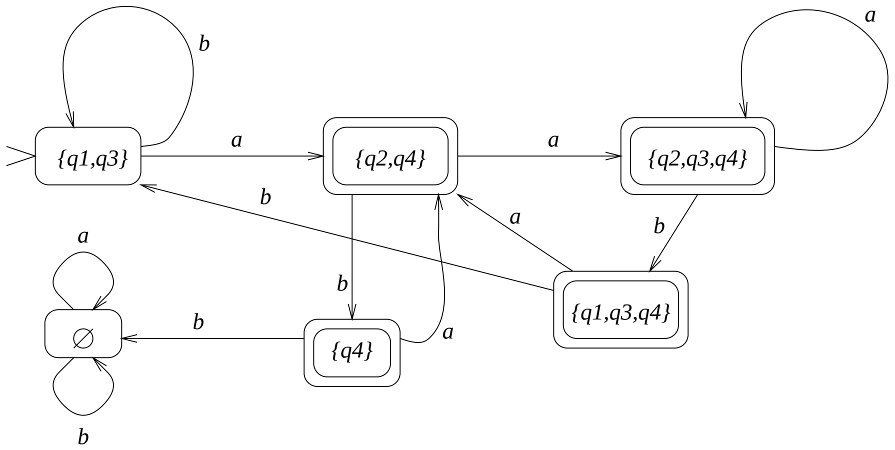

Figure 5: Result of the powerset-construction

power set of K), s' is the set of all states in M' reachable from s using only ϵ-transitions:

$$s' = \{q \mid (s, \epsilon) \vdash^* (q, \epsilon)\}$$

F' is the set of all subsets of K containing at least one final state from M:

$$F' = \{Q \subseteq K \mid Q \cap F \neq \emptyset\}$$

and δ represents the transition function of the new automaton given by:

$$\delta(Q, a) = \{q' \mid \exists q \in Q[(q, a) \vdash^* (q', \epsilon)]\}$$

(where by $\vdash$ we refer to the original automaton M).

An example is given in Figure 5. Not all states are given: those that are not reachable from the new initial state $s' = \{q1, q3\}$ have been omitted.

The existence of the above powerset-construction implies that for any nondeterministic finite automaton, there is a deterministic finite automaton accepting the same language. Since it is trivial to formulate a construction the other ways around (a deterministic finite automaton can be seen as a special kind of nondeterministic finite automaton) we now know that both types of finite automaton are equivalent, with regard to the kind of language that they can accept. However, a nondeterministic finite automaton can be more succinct than a deterministic one.

Sometimes, a finite automaton is described using a transition *relation*, as opposed to a transition *function*, yet no actual nondeterminism occurs, i.e. in no configuration more than one transition is applicable. In that case, we may treat the automaton as deterministic, although strictly speaking it does not satisfy the mathematical definition of deterministic finite automata.

5 The relation between regular languages and finite automata

Any language described by a finite automaton (be it deterministic of nondeterministic) can also be described by a regular expression and vice versa. In other words, the regular

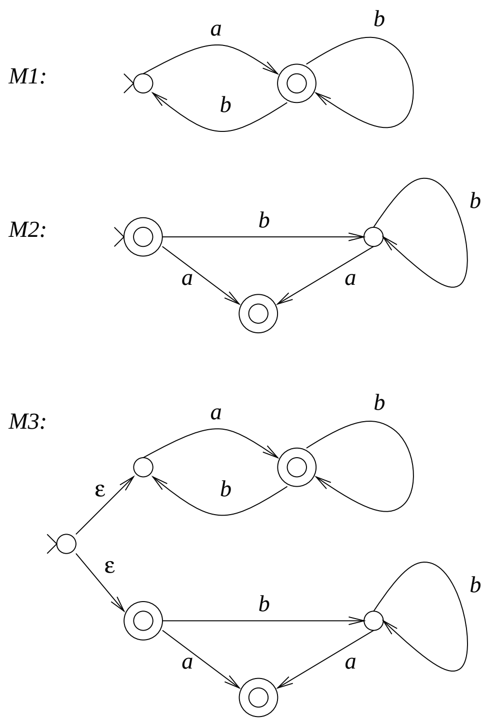

Figure 6: Union of regular languages. (We construct M_3 from M_1 and M_2 by merging the states and transitions from M_1 and M_2, and introducing a new initial state.)

languages can also be defined to be the class of languages accepted by (non)deterministic finite automata.

We will not prove this (see e.g. [LP81] for a proof) but we will argue in detail that some properties hold both for regular expressions and for finite automata.

For example, that the regular languages are closed under union (i.e. if L_1 and L_2 are regular, then also $L_1 \cup L_2$ is regular), can be argued by mentioning that $\cup$ is an operator that may occur in regular expressions, but also by giving a construction of one finite automaton from two others. In particular, suppose that we have two finite automata M_1 and M_2 accepting languages L_1 and L_2, respectively, then we can construct a third automaton M_3 which accepts the language $L_1 \cup L_2$. Figure 6 gives an example. Similarly, we can construct an automaton accepting the concatenation of two regular languages (Figure 7). A third construction shows that also the Kleene star operator can be expressed as the construction of one finite automaton from another: if M_1 accepts some language L, then M_2 accepts the language L^*. See Figure 8 for an example.

The general constructions can be found in [LP81].

An interesting property which does not follow immediately from the structure of regular expressions or finite automata is that regular languages are closed under intersection. I.e. if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$. The proof can be given by constructing a finite automaton M_3 from two automata M_1 and M_2, much as above. The construction is a little more complicated however.

Suppose that we have $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (K_2, \Sigma, \delta_2, s_2, F_2)$.[5] The new

[5]Without loss of generality, we assume the automata have the same alphabet and are deterministic.

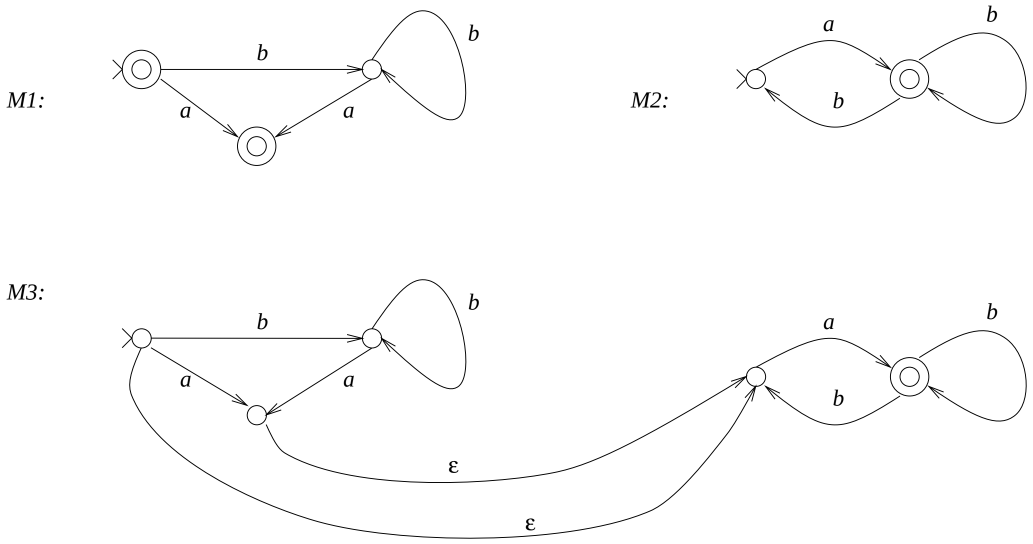

Figure 7: Concatenation of regular languages. (We construct M_3 from M_1 and M_2 by merging the states and transitions from M_1 and M_2, and connecting the former final states from M_1 to the former initial state of M_2.)

automaton M_3 that we construct simulates M_1 and M_2 simultaneously. Its states are pairs, consisting of a state that M_1 would be in, and one that M_2 would be in. Formally, $M_3 = (K_1 \times K_2, \Sigma, \delta_3, (s_1, s_2), F_1 \times F_2)$, where we define δ_3 by

$$\delta_3((q_1, q_2), a) \quad = \quad (\delta_1(q_1, a), \delta_2(q_2, a))$$

for all $q_1 \in K_1$, $q_2 \in K_2$, and $a \in \Sigma$. See Figure 9 for an example.

Another property which can be easily proven by a construction of finite automata is closure of regular languages under complementation: suppose we take a regular language, described by a deterministic finite automaton, and we make all final states non-final, and all formerly non-final states final. Then the resulting automaton accepts exactly those strings that the old automaton did not accept. Therefore, if L is regular, then so is $\overline{L}$.

6 Rational relations

Rational relations have much in common with regular languages. A regular relation can also be described using a regular expression or a finite automaton, but with a special alphabet, consisting of pairs (v_1, v_2), where v_1 is a string over an *input alphabet* Σ_1 and v_2 is a string over an *output alphabet* Σ_2.

For example, suppose that the input and output alphabets are $\Sigma_1 = \Sigma_2 = \{a, b\}$. We can make a regular expression describing a rational relation between strings in Σ_1^* and Σ_2^* as

$$(\ (a, a) \cup (b, b)\ )^*\ (aba, aa)\ (\ (a, a) \cup (b, b)\ )^*$$

An example of a string generated by this expression is $(a, a)(b, b)(aba, aa)(b, b)$. If we concatenate all first components in the pairs, we obtain *ababab* and if we concatenate all second components then *abaab*. The difference is that an occurrence of b between two a's

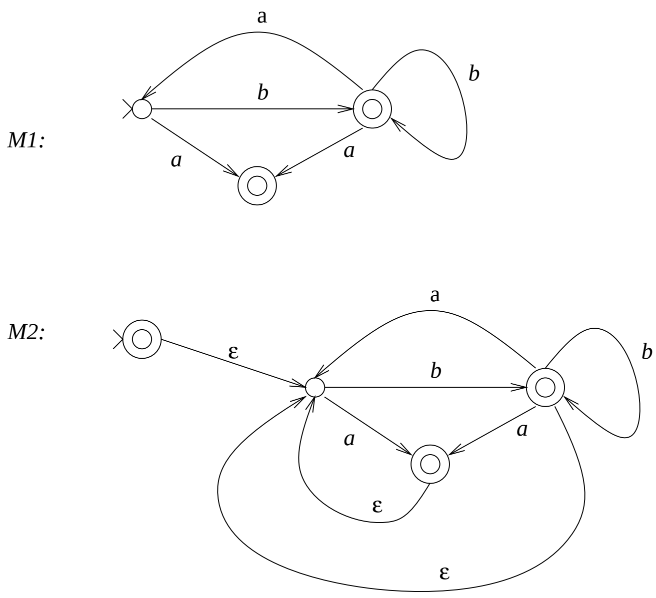

Figure 8: Kleene star applied to a regular language. (We construct M_2 from M_1 by introducing a new initial state, which is also final. Further, from old final states epsilon transitions lead back to the former initial state.)

has been removed. In general, the above regular expression describes a rational relation between Σ_1^* and Σ_2^* given by: v is in relation to w if v contains a b between two a's, and w is identical, except that the b has been removed.

Formally, if we have a regular language L over $\Sigma_1^* \times \Sigma_2^*$, then this describes the binary relation R defined by: $v \ R \ w$ if and only if there is a sequence $(v_1, w_1) \cdots (v_m, w_m) \in L$, some m, such that $v = v_1 \circ \ldots \circ v_m$ and $w = w_1 \circ \ldots \circ w_m$. If we want to see binary relations as sets of pairs, then we can interpret R alternatively as a set $\subseteq \Sigma_1^* \times \Sigma_2^*$ defined by

$$R = \{(v_1 \circ \ldots \circ v_m, w_1 \circ \ldots \circ w_m) \mid (v_1, w_1) \cdots (v_m, w_m) \in L\}$$

By definition, we call such a relation a *rational* relation. Note that since we can describe regular languages by both regular expressions and finite automata, we can also describe rational relations by means of regular expressions over the product alphabet $\Sigma_1^* \times \Sigma_2^*$ or by means of a finite automaton of which the arrows are labelled by pairs of strings. Such a finite automaton is then called a *finite transducer*. Usually, we label the arrows of finite transducers with single elements from $\Sigma_1^* \times \Sigma_2^*$.[6]

For the running example, we have the alternative formulation of the rational relation by means of the finite transducer in Figure 10. Instead of using pairs (v, w), we label the arrows in the transducer with expressions of the form v/w.

The class of rational relations is closed under concatenation, union, and Kleene star. This directly follows from the corresponding closure properties of regular languages. Surprisingly however, rational relations are *not* closed under intersection. For example, con-

[6]Instead of an arrow labelled with $(v_1, w_1) \cdots (v_m, w_m) \in (\Sigma_1^* \times \Sigma_2^*)^*$, $m \neq 1$, we may as well label the arrow with a single tuple $(v_1 \cdots v_m, w_1 \cdots w_m)$.

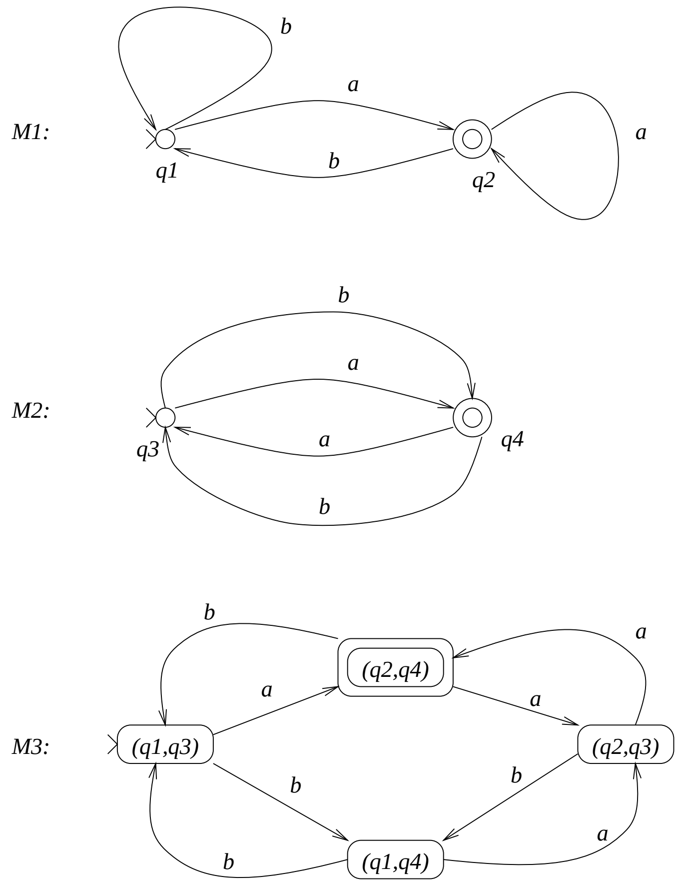

Figure 9: The intersection of two regular languages.

sider the rational relations given by[7]

$$\begin{aligned} R_1 &= (a,b)^*(\epsilon,c)^* = \{(a^n, b^n c^m) \mid n,m \in \{0,1,2,\ldots\}\} \\ R_2 &= (\epsilon,b)^*(a,c)^* = \{(a^n, b^m c^n) \mid n,m \in \{0,1,2,\ldots\}\} \end{aligned}$$

The intersection is

$$R_1 \cap R_2 = \{(a^n, b^n c^n) \mid n \in \{0,1,2,\ldots\}\}$$

which can be shown not to be rational [Ber79, p. 58].

For finite automata, transitions with strings $v \in \Sigma^*$, $|v| > 1$, are not really essential for description of regular languages. Similarly, finite transducers can do without transitions that read or write more than one symbol at a time. The formal statement is that any rational relation can be described by a finite transducer which only has labels v/w, where $v \in \Sigma_1 \cup \{\epsilon\}$ and $w \in \Sigma_2 \cup \{\epsilon\}$.

Note that a rational relation can be turned into a regular language by omitting either the input or output components. For example, by omitting the input components in $(a,b)^*(\epsilon,c)^*$ one obtains b^*c^*, a regular language. This observation implies that each

[7] A superscript n, as in a^n, means the concatenation of n occurrences of a string. E.g. $a^3 = aaa$.

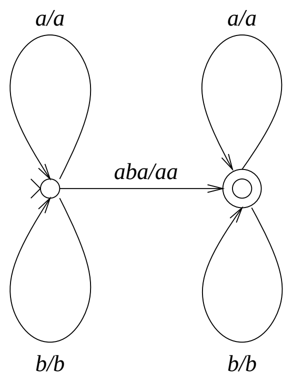

Figure 10: A finite transducer

rational relation relates strings in two regular languages, the *input language* and the *output language*. For Figure 10, the input language is the set of all strings over $\{a, b\}$ containing at least one occurrence of aba, and the output language is the set of all strings over $\{a, b\}$ containing at least one occurrence of aa.

7 Rational functions

A *rational function* is a special kind of rational relation, where each input string is related to at most one output string. Formally, a rational relation R is a rational function if for all v, the set $\{w \mid (v, w) \in R\}$ has either zero or one elements. Note that a rational function is in general a *partial* function, since $\{w \mid (v, w) \in R\}$ may contain zero elements for some $v \in \Sigma_1^*$.

For example, the rational relation given by the regular expression

$$(xx, aa)^* \quad \cup \quad (x, b)(xx, bb)^*$$

is a rational function that maps a string of x's to a string of the same length of a's if the length of the string is even. If the length is odd, then the output is a string of b's of the same length.

On the other hand, the rational relation

$$(xxx, aaa)^* \quad \cup \quad (xx, bb)^*$$

is *not* a function, since a string x^n, where n is a multiple of 6, is related to both a^n and b^n by the rational relation.

In [Ber79, p. 94] it is proved that it is decidable whether a rational relation is a function, by taking a finite transducer and investigating the transductions for a finite number of strings.

Related to rational functions are *unambiguous* (finite) transducers. We call transducers unambiguous if for any input string, there is at most one path through the automaton from the initial to a final state for that input string. Of course, any unambiguous transducer represents a rational function (for that at most one path for a given input string, there is only one output string). Conversely, for any rational function there is an unambiguous transducer (see [Ber79, p. 115] for the proof).

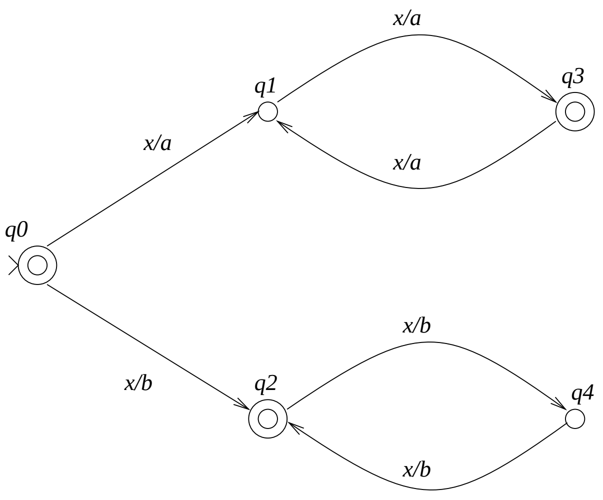

Figure 11: An unambiguous transducer (adopted from [Ber79])

For example, for the rational function described by $(xx, aa)^* \cup (x, b)(xx, bb)^*$, we can construct the unambiguous transducer from Figure 11.

Unambiguous transducers are closer to implementable algorithms than arbitrary (finite) transducers. Yet, there may still be nondeterminism while processing an input string and computing the output string: even if there can be only one path through the automaton leading to a final state when the input has been read completely, during intermediate steps one may need to hypothesize alternative paths through the automaton.

This is illustrated by Figure 11. While processing the string of x's, one needs to hypothesize about both the paths leading through *q1* and *q3* and those leading through *q2* and *q4*. Only upon finding the end of the string can it be determined which is the correct path.

8 Sequential transducers

In this section we discuss a type of automaton which can be directly implemented on a computer. Formally, a *sequential transducer* is a 5-tuple $M = (K, \Sigma_1, \Sigma_2, \delta, s)$, where K is a finite set of *states*, of which s is the *initial state*, Σ_1 and Σ_2 are the input and output alphabets, respectively, and δ is a partial function from $K \times \Sigma_1$ to $K \times \Sigma_2^*$. A sequential transducer can be seen as a finite transducer of which all states are final, and that is deterministic, i.e. for any present state and next input symbol there is at most one transition to another state, and in this transition only one output string can be written.

Formally, the initial configuration is (s, v, ϵ), where v is the input string. The relation $\vdash$ is defined by: $(q, av, w) \vdash (q', v, wx)$ if $\delta(q, a) = (q', x)$.

A sequential transducer describes a partial function, called *sequential function*. Formally, a string v is mapped to a string w if $(s, v, \epsilon) \vdash^* (q, \epsilon, w)$, some state q.

Consider for example the sequential transducer given in Figure 12, with $\Sigma_1 = \{x, y\}$ and $\Sigma_2 = \{a, b\}$. (That $\delta(q, a) = (q', x)$ is indicated by an arrow from a node representing q to one representing q' labelled a/x.) This function maps a string v over Σ_1 of length n to a^n if v starts with an x, and maps it to b^n otherwise.

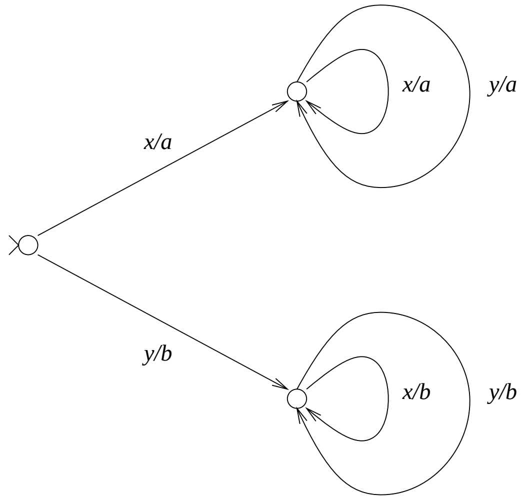

Figure 12: A sequential transducer (adopted from [Ber79])

The definition of sequential transducers above is biased towards left-to-right processing. Therefore, sequential transducers are also called more specifically *left* sequential transducers. This implies that there are also *right* sequential transducers, which process input from right-to-left. Here, the configurations contain prefixes of the input, and $\vdash$ is defined by: $(q, va, w) \vdash (q', v, xw)$ if $\delta(q, a) = (q', x)$.

A left sequential function (i.e. a function defined by means of a left sequential transducer) is not necessarily also a right sequential function. E.g. the left sequential function from Figure 12 cannot be described by any right sequential transducer. (Informally, this can be shown by the argument that some right sequential transducer would have to start writing either a's or b's for each of the symbols it reads in the input from right-to-left, before it can have any knowledge whether the left-most symbol of the input is an x or y.)

Any (left or right) sequential function is a rational function. However, not each rational function is sequential. For example, the rational function specified in Figure 11 is not a sequential function.

A particular property of (left) sequential functions is that if w is mapped to v and if wx is mapped to some string v', then v' is of the form vy, some y. This property of called *preservation of left factors.* In a similar way, right sequential functions preserve right factors.

9 Subsequential transducers

A slight extension of the sequential functions are the *subsequential* functions described below.

A subsequential transducer is a 6-tuple $M = (K, \Sigma_1, \Sigma_2, \delta, \lambda, s)$, where λ is a partial function from K to Σ_2^*, and the other components are as in the definition of sequential transducers.

The purpose of the new component λ is to place an additional string after the output string when the end of the input string has been reached. Formally, an input v is mapped to an output w by the subsequential function if $(s, v, \epsilon) \vdash^* (q, \epsilon, w')$, $\lambda(q) = x$ and $w = w'x$.

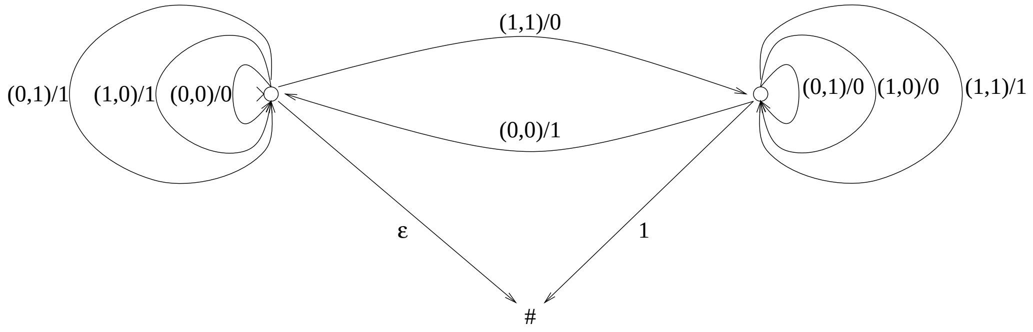

Figure 13: A subsequential transducer for addition (adopted from [Ber79])

Any sequential function is a subsequential function (take $\lambda(q) = \epsilon$ for all $q \in K$), but not each subsequential function is a sequential function. An example where detection of the end of the input is necessary, and where sequential transducers do not suffice is binary addition, described as follows.

Suppose we have two strings $a_1 \cdots a_n$ and $b_1 \cdots b_n$ over $\{0, 1\}$ of the same length. These strings represent two binary numbers. We want to compute the addition $zc_1 \cdots c_n$, where z is ϵ or 1, by means of a subsequential transducer. As input string we take $(a_n, b_n) \cdots (a_1, b_1) \in (\{0, 1\} \times \{0, 1\})^*$. We compute also the binary addition in reverse form, i.e. $c_n \cdots c_1 z$, in the output string.

The subsequential transducer has two states presenting the "carry" from the addition of the previous two "bits". When all bits from the input have been seen, then the remaining carry should be written to the output and for this we need λ. Our solution is given in Figure 13. We indicate $\lambda(q) = w$ by an arrow labelled w from q to the symbol $\#$.

The class of subsequential functions is properly contained in the class of rational functions. It is decidable whether a rational relation is subsequential, and also whether it is furthermore sequential [Ber79, p. 128].

Intuitively, subsequential transducers relate to finite transducers as deterministic finite automata relate to nondeterministic finite automata. However, the powerset-construction of finite automata does not carry over in a trivial way to transducers, which is apparent when we consider that the subsequential functions are *properly* contained in the rational relations. However, a construction in [RS95], which is related to the powerset-construction, is capable of deriving subsequential transducers from those finite transducers that describe subsequential functions.

Two (sub)sequential functions f from Σ_1^* to Σ^* and g from Σ^* to Σ_2^* can be composed into a partial function $g \circ f$ from Σ_1^* to Σ_2^*, defined by $(g \circ f)x = g(f(x))$, for all $x \in \Sigma_1^*$.[8] In other words, the output of f is input to g.

The composition of two (sub)sequential functions is again (sub)sequential. In the following section we investigate what happens if a left sequential transducer is composed with a right sequential transducer.

[8]The composition operator $\circ$ should not be confused with the operator of the same name that is sometimes used to denote concatenation.

10 Bimachines

A *bimachine* is a 9-tuple $M = (K^l, K^r, \Sigma_1, \Sigma_2, \delta^l, \delta^r, s^l, s^r, \gamma)$, where K^l and K^r are two sets of states, Σ_1 and Σ_2 are the input and output alphabets, as before, δ^l is a function from $K^l \times \Sigma_1$ to K^l, δ^r is a function from $K^r \times \Sigma_1$ to K^r, $s^l \in K^l$ is the left-initial state and $s^r \in K^r$ is the right-initial state, and γ is a partial function from $K^l \times \Sigma_1 \times K^r$ to Σ_2^*.

A bimachine operates simultaneously from left-to-right and from right-to-left. Starting with s^l, states in K^l are computed at each of the input positions from left-to-right by δ^l. Conversely, starting with s^r, states in K^r are computed at each of the input positions from right-to-left by δ^r.

Suppose the input is given by $a_1 \cdots a_n \in \Sigma_1$. We compute the states $l_1, \ldots, l_n \in K^l$ and $r_1, \ldots, r_n \in K^r$ by

$$
\begin{aligned}
l_1 &= s^l \\
l_{m+1} &= \delta^l(l_m, a_m), \text{ for } 1 < m \leq n \\
r_n &= s^r \\
r_{m-1} &= \delta^r(r_m, a_m), \text{ for } 1 \leq m < n
\end{aligned}
$$

The output string is now defined by the concatenation $\gamma(l_1, a_1, r_1) \circ \cdots \circ \gamma(l_n, a_n, r_n)$, provided γ is defined on all its argument in this expression; otherwise the transduction on input $a_1 \cdots a_n$ is not defined.

As an example, consider the function f from Σ_1^* to Σ_2^*, with $\Sigma_1 = \{x, y\}$ and $\Sigma_2 = \{a, b\}$, defined by

$$
\begin{aligned}
f(w) &= a^{|w|}, \text{ if } w = xw'x \text{ some } w' \\
&= b^{|w|}, \text{ otherwise}
\end{aligned}
$$

In words, a string is mapped to a string of the same length, consisting of a's if the input has x's at its two ends, and consisting of b's otherwise.

We can build a bimachine for this problem by taking $K^l = \{\perp, 0, 1\}$, $K^r = \{\perp, 0, 1\}$, $s^l = \perp$, $s_r = \perp$, and δ^l and δ^r such that

$$
\begin{aligned}
\delta^l(\perp, x) &= 1 \\
\delta^l(\perp, y) &= 0 \\
\delta^l(1, z) &= 1, \ z \in \Sigma_1 \\
\delta^l(0, z) &= 0, \ z \in \Sigma_1 \\
\delta^r(\perp, x) &= 1 \\
\delta^r(\perp, y) &= 0 \\
\delta^r(1, z) &= 1, \ z \in \Sigma_1 \\
\delta^r(0, z) &= 0, \ z \in \Sigma_1
\end{aligned}
$$

and γ such that

$$
\begin{aligned}
\gamma(\perp, x, 1) &= a \\
\gamma(1, x, \perp) &= a \\
\gamma(1, z, 1) &= a, \ z \in \Sigma_1 \\
\gamma(p, z, q) &= b, \text{ for all other arguments}
\end{aligned}
$$

For input $xyxx$ we obtain the states $\perp, 1, 1, 1 \in K^l$ and $1, 1, 1, \perp \in K_r$. The output is computed by $\gamma(\perp, x, 1)\gamma(1, y, 1)\gamma(1, x, 1)\gamma(1, x, \perp) = aaaa$. For input $yxyx$ we obtain $\perp, 0, 0, 0 \in K^l$ and $1, 1, 1, \perp \in K_r$, in which case the output is computed by $\gamma(\perp, y, 1)\gamma(0, x, 1)\gamma(0, y, 1)\gamma(0, x, \perp) = bbbb$.

Any rational function can be described by a bimachine, provided this function maps ϵ to ϵ. (Bimachines cannot map ϵ to a non-empty string.) Conversely, any bimachine describes a rational function. For a proof, see [Ber79, p. 125].

If again we restrict ourselves to functions that map ϵ to ϵ, then bimachines represent exactly the functions that can be described by means of the composition of a left sequential function and a right sequential function.[9] Proof can be found in [Ber79, p. 126]. Here we just demonstrate that the bimachine of the running example can also be described as the composition of a left sequential function and a right sequential function.

We define the left sequential transducer by $M^l = (\{\perp, 0, 1\}, \Sigma_1, \{\perp, 0, 1\} \times \Sigma_1, \delta^l, \perp)$, where δ^l is defined by

$$\begin{array}{rcl}
\delta^l(\perp, x) & = & (1, (\perp, x)) \\
\delta^l(\perp, y) & = & (0, (\perp, y)) \\
\delta^l(1, z) & = & (1, (1, z)),\ z \in \Sigma_1 \\
\delta^l(0, z) & = & (0, (0, z)),\ z \in \Sigma_1
\end{array}$$

Application of M^l to $xyxx$ yields $(\perp, x)(1, y)(1, x)(1, x)$, and application to $yxyx$ yields $(\perp, y)(0, x)(0, y)(0, x)$.

We define the right sequential transducer by $M^r = (\{\perp, 0, 1\}, \{\perp, 0, 1\} \times \Sigma_1, \Sigma_2, \delta^r, \perp)$, where δ^r is defined by

$$\begin{array}{rcl}
\delta^r(\perp, (1, x)) & = & (1, a) \\
\delta^r(\perp, (q, x)) & = & (1, b),\ q \in \{\perp, 0\} \\
\delta^r(\perp, (q, y)) & = & (0, b),\ q \in \{\perp, 0, 1\} \\
\delta^r(1, (\perp, x)) & = & (1, a) \\
\delta^r(1, (\perp, y)) & = & (1, b) \\
\delta^r(1, (1, z)) & = & (1, a), z \in \Sigma_1 \\
\delta^r(1, (0, z)) & = & (1, b), z \in \Sigma_1 \\
\delta^r(0, (q, z)) & = & (0, b), z \in \Sigma_1, q \in \{\perp, 0, 1\}
\end{array}$$

Application of M^r to $(\perp, x)(1, y)(1, x)(1, x)$ yields $aaaa$ and application to $(\perp, y)(0, x)(0, y)(0, x)$ yields $bbbb$.

Generalisation of this example results in the above-mentioned proof that any bimachine is the composition of a left sequential function and a right sequential function.

For a practical application of bimachines, see [Roc94].

11 Suggestions for further reading

A good introduction to finite automata and regular languages can be found in [HU79, Chapters 2 and 3], [LP81, Chapters 1 and 2] and [Har78, Chapters 1 and 2]. A few definitions regarding finite transducers can be found in [Gur89]. One of the most thorough

[9] Or conversely as the composition of a right sequential function and a left sequential function.

books on finite-state transductions and related topics is [Ber79], which is however difficult to read for students without sufficient mathematical background.

References

[Ber79] J. Berstel. *Transductions and Context-Free Languages.* B.G. Teubner, Stuttgart, 1979.

[Gur89] E.M. Gurari. *An Introduction to the Theory of Computation.* Computer Science Press, 1989.

[Har78] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[Roc94] E. Roche. Two parsing algorithms by means of finite state transducers. In *The 15th International Conference on Computational Linguistics*, volume 1, pages 431–435, Kyoto, Japan, August 1994.

[RS95] E. Roche and Y. Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, 1995.