1 Introduction

At some level all mechanical computing is symbol processing. And, at least in current architectures, all these symbols are arranged as strings – linear lists. So it makes sense to turn to the problem of representing sets of strings, or *languages* and characterizing languages of particularly simple or useful forms.

Informally, strings are sequences of symbols. Symbols come from some finite alphabet. The main operation on strings is concatenation. Languages are simply sets of strings, frequently recursively defined.

2 Strings

An *alphabet* is any finite set, and is usually denoted Σ. The most common alphabet we work with will be $\{a, b\}$, though others will certainly appear.

We will give two formal definitions of *string* or *word* over Σ. Intuitively, a string is a sequence of symbols of some finite length. So, the first definition represents a string as a function:

$$f : \{0, 1, 2, \ldots, n-1\} \to \Sigma$$

Remembering our definition of $\mathbb{N}$ we could also write this as simply $f : n \to \Sigma$. So, the string aab is represented by the function $f : 3 \to \{a, b\}$ with $f(0) = f(1) = a$, $f(2) = b$. The fundamental operation on strings is concatenation e.g. the concatenation of aab with ba is $aabba$. In the functional representation of strings, the concatenation of $f : n \to \Sigma$ with $g : m \to \Sigma$, is $f \cdot g : n + m \to \Sigma$ where:

$$(f \cdot g)(k) = \begin{cases} f(k) & \text{if } k < n \\ g(k-n) & \text{if } n \leq k \end{cases}$$

If $f : n \to \Sigma$ is a string, we say its *length* is n and write $|f| = n$. So $|f \cdot g| = |f| + |g|$. There is a unique string of length 0 (because there is a unique function whose domain is the empty set), and it is denoted ϵ. Obviously $f = \epsilon \cdot f = f \cdot \epsilon$ for any string f. The set of all strings over Σ is denoted Σ^*.

Note that we can also represent strings recursively using a linked list.

Recursive definition of Σ^*
Base: $\epsilon \in \Sigma^*$ (where ϵ represents the empty string)
Construction: If $w \in \Sigma^*$ and $x \in \Sigma$, then $(w, x) \in \Sigma^*$.

With the recursive definition, how do we define concatenation? Recursively of course:

$$\begin{aligned} u \cdot \epsilon &= u \\ u \cdot (w, x) &= (u \cdot w, x) \end{aligned}$$

And likewise we need to define length recursively (the length of the empty string is 0 and the construction step increases length by 1).

From now on we'll generally ignore the formality and assume "string", "concatenation" and "length" as primitive concepts. Furthermore we usually drop the $\cdot$ in concatenations.

3 Languages

A *language* (over Σ) is any subset of Σ^*. We extend the notion of concatenation of strings to languages, namely if L and K are languages over Σ^* we define:

$$LK = \{uv \mid u \in L, v \in K\}.$$

Here are some simple examples of languages over $\{a, b\}$:

START-a: All strings that start with a. In the functional definition, this means all functions $f : n \to \Sigma$ for some $n > 0$, with $f(0) = a$. Or we could just represent it as a concatenation $\{a\}\Sigma^*$. Or recursively: $a \in$ START-a, and if $w \in$ START-a and $x \in \Sigma$ then $wx \in$ START-a.

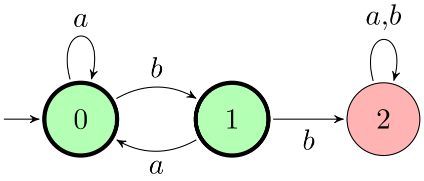
Let $T = \{aa, ab, ba, bb\}$, and define a language ES recursively as follows: $\epsilon \in ES$, and if $w \in ES$ and $v \in T$ then $wv \in ES$. Of course we claim that ES consists of all strings of even length. As practice, let's prove that claim by induction. There are two things we need to check: that every string in ES has even length, and that every string of even length is in ES.

For the first part, note that ϵ has length 0 which is even. Also if the length of w is even and $u \in T$ then $|wu| = |w| + 2$ is also even. Hence, by induction, every string in ES has even length.

For the second part, suppose that $w \in \Sigma^*$ has even length. If $|w| = 0$ then $w = \epsilon \in ES$. Otherwise $w = u \cdot v$ for some u of even length and v of length 2. By induction we may assume that $u \in ES$, and since $v \in T$, we get $w \in ES$. So, every string of even length is in ES and we're done.

4 Finite state automata

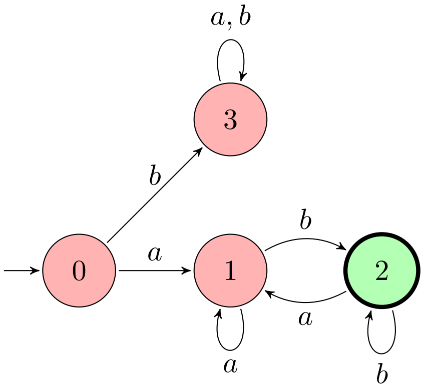
We're now going to talk about some "computing machines". But, these machines are very limited compared to the general purpose computers we are used to. In fact, they are much more similar to various hardware controlled units (old fashioned adding machines, vending machines, typewriters). The basic idea is that the machine is a black box with a key board attached, and a light. Certain combinations of key strokes cause the light to shine – what's actually going on is that there are finitely many internal states possible, and the various key strokes cause transitions between those states. Some of the states are marked as "accepting" and if we are in one of those states, then the light shines. Associated to the machine is a language that it accepts – those sequences of key strokes which, from the initial state, cause the light to be on. The challenge is to model all this and to determine what sorts of languages are accepted by these finite state automata.



In the diagram above, the little $>$ symbol points at the initial state, and the arrows indicate the state transitions caused by various inputs. The green circles represent accepting or "light on" states, while a red circle represents a rejecting state. It's easy to see that once we get to the rejecting state we stay there and we get there exactly if we see two b's in a row in the input. So, this machine recognizes the language HAS-NO-bb.

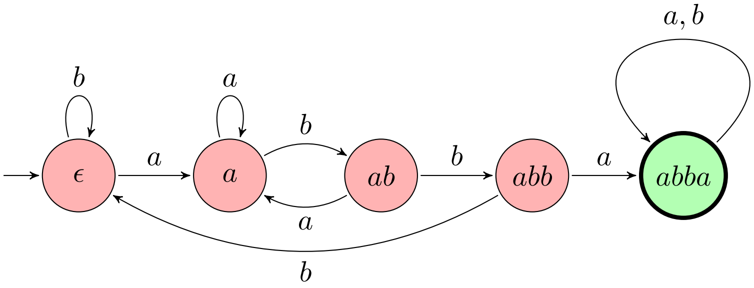
5 More examples

The following machine recognizes the language "starts with a and ends with b"



Note again that we enter a "cannot succeed" state (3) if we begin with b. The light is on at (2) whenever we started with a and finished with b – it stays on if we weren't actually finished typing but type some more b's. It turns off again if we add another a.

Now consider the language HAS-$abba$ of strings that contain $abba$ as a consecutive substring. This time to be helpful we label the states. The labels indicate how long a prefix of $abba$ we have seen in the last few characters of the input. Once we reach 4 of course the light goes on and stays on. This time we don't need a "cannot succeed" state because there's always hope of seeing $abba$ in the future.



6 Deterministic finite state automata

A *deterministic finite state automaton*, M, consists of:

- A finite set, $\mathcal{S}$, of *states*;
- A finite alphabet Σ of *actions*;
- A total function $T : \mathcal{S} \times \Sigma \to \mathcal{S}$ called the *transition function*
- A distinguished state $s \in \mathcal{S}$ called the *initial state*; and
- A subset A of Q called the *accepting* states.

The letters chosen don't mean anything but are supposed to be a bit mnemonic and differ from the "standard" choices you may see elsewhere (Q for states, δ for the transition function, q_0 for the initial state, and F for the accepting states).

Given a word $w = w_0 w_1 \dots w_{n-1} \in \Sigma^*$ the *computation* carried out by M on input w is a sequence of states $s_0, s_1, s_2, \dots, s_n$ defined as follows:

$$s_0 = s, s_1 = T(s_0, w_0),\ s_2 = T(s_1, w_1),\ \dots,\ s_n = T(s_{n-1}, w_{n-1})$$

We say that M *accepts* or *recognises* w if $s_n \in A$ and otherwise it *rejects* w.

The *language of* M, $L(M)$ is just the set of strings in Σ^* that M accepts.

7 Non determinism

We can loosen the definition above to allow several forms of non-determinism in finite state automata.

The first of these is akin to keyboard jam – instead of demanding that T be a total function from $\mathcal{S} \times \Sigma$ to $\mathcal{S}$ we allow it to be partial. This allows us to eliminate "can't accept" states, by simply not including any transitions that would lead to such states.

The other forms of non-determinism extend rather than restrict the transition function. In "multiple worlds" (not an official name) non-determinism, we simply allow T to be a relation on $\mathcal{S} \times \Sigma \times \mathcal{S}$. For each triple $(m, x, n) \in T$ we might transition from m to n on symbol x. But equally there might be some other $(m, x, o) \in T$ and we could go to o instead. Now, instead of a single possible computation on input w we have a whole range of possibilities – basically we just require that $(s_i, w_i, s_{i+1}) \in T$. We define $L(M)$ in this case to be the set of words for which *some* computation winds up in F.

Finally, we could allow ϵ transitions, a.k.a. "bumping the machine" (also not official). Here we allow some transitions $m \to n$ which do not consume any input (i.e. occur on a symbol ϵ). We accept w if there is some way of interspersing ϵ's among its elements and then following an accepting computation.

8 Exercises

"If you build it, you will have built it."

1. In each part of this question, design a finite automaton that accepts the given language. As usual, take the alphabet to be $\{a, b\}$.

   (a) The language consisting of all strings of the form "0 or more a's followed by 0 or more b's".

   (b) The language of all strings that contain exactly two a's.

   (c) The language EvenString consisting of all strings of even length.

   (d) The language consisting of all strings not containing the substring bb.

   (e) The language $\mathsf{Even} - \mathsf{even}$ consisting of all strings containing both an even number of a's and an even number of b's.

2. For each of the following languages, try to design a finite automaton that accepts it – but don't try for too long, because you can't. Try to get a feeling for what the problem is.

   (a) The language of all strings of the form $a^n b^n$ $(n \geq 0)$.

   (b) The language, L of all strings over the alphabet $\{\langle, \rangle\}$ described recursively by:

   $$\begin{array}{l} \epsilon \in L, \text{and} \\ \text{if } u, v \in L \text{ then } \langle u \rangle v \in L. \end{array}$$

   (these are called *balanced bracket sequences*).

3. In each part of this question, design an NFA that accepts the given language. Try to make it as small as possible, and compare to a DFA for the same language.

   (a) The language of all strings ending with b.

   (b) The language of all strings containing the substring bb.

   (c) The language of all strings containing either the substring bb or the substring $baab$.