Languages, Automata and Computation

Thomas Forster

May 20, 2005

Contents

Introduction

Mathematical background

I assume that the reader knows what a natural number is.

There is a genuine hard-core proof by induction (Kleene's theorem) but it is not examinable.

There are some ideas from Logic 122 that readers will find helpful: it will help them if they know what conjunctive and disjunctive normal form are tho' it's not necessary; Universal Generalisation and $\forall$-introduction would help too, but chiefly for Kleene's theorem which is not examinable. Congruence relations are mentioned on page 11. Disjoint unions appear too.

We will be using the asterisk symbol: *, in more than one way. We will do the same with the two vertical bars ||. When a symbol is used in two related ways, we say it is **overloaded**.

Useful URLs

There are many of course. Here are three that have been recommended to me.

- I've tutored courses using the material on

  `http://www.cl.cam.ac.uk/Teaching/2002/RLFA/reglfa.ps.gz`

- My colleague Richard Crouch has some useful teaching material on his home page. Look under `http://www.parc.com/crouch`. Under "Teaching Material" you will find pdf files for "Formal Language Theory".

- Dr Wu recommends JFLAP. It is a complete package for doing almost anything relating to fsa, cfg, tm, pda, L-system etc etc. Go to `http://www.cs.duke.edu/~rodger/`

Chapter 1

Machines

This is a course on Languages, Automata and Computation. We have to start somewhere, and 'a' is before 'c' or 'l' so let's start with automata!

'Automata' (singular: automaton) is a Greek word for 'machines'. The automata in this course are all *discrete* rather than *continuous* machines. Or perhaps one should say *digital*-rather-than-*analogue*. If you are not happy about this difference go away and ask your tutor before you read any further. If you are happy about this difference just check that you and I have the same take on it: a car is an *analogue* machine, and a computer is a *digital* machine.

There are two ways into machines, and one can profitably run both—one does not have to choose one.

- One can draw a couple of FSAs and talk through what they do. To this end,

  `click on this animation`

  http://people.pwf.cam.ac.uk/rmn30/fsm/fsmApplet.html

- One can start with a well-motivated story from life and say how this might be a machine. You may remember the water jugs problem from coursework 1.

  Try for example:

  `http://www.philosophy.uncc.edu/logic/projectTALLC/tallc2/applets/puzzles/wj/wj_jar.html`
  or try googling 'water jugs'

snakes and ladders

Finite state machines can be quite useful in describing games, like chess, or Go, or draughts. This is because many games (for example, those I have just mentioned) have machines naturally associated with them: the board positions can be thought of as states of a machine. It's not always clear what the input alphabet is!

The point of departure is this. machines are things with finite descriptions that have states, and they move from one state to another on receiving an input, which is a character from an input alphabet.

To be formal about it: A machine $\mathcal{M}$ is a set S of states, together with a family of transition operations, one for each $w \in \Sigma$, the input alphabet. These transition operations are usually written as one function of two arguments rather than lots of unary operations. Thus $\sigma(s, c)$ is the state that machine is in after it received character c when it was in state s:

$$\sigma : S \times \Sigma \to S$$

There must of course be a designated start state $s_0 \in S$, and there is a set A of accepting states $A \subseteq S$, whose significance we will explain later. We will also only be interested in machines with only finitely many states. There are technicalities to do with infinity, but we don't need to worry about them just yet. All we mean is that for our machine $\mathcal{M}$ the number $|S|$ (the size of the set of states) must be a natural number: $|S| = 1$ or $|S| = 2$ or ....

1.1 Languages recognised by machines

Languages, parsing, compilers etc are the chief motivation for this course. You might think that a *language* is something like English, or Spanish, or perhaps (since this is a computer Science course) PASCAL or JAVA or something like that: a naturally occurring set of strings-of-letters with some natural definition and a sensible purpose such as meaning something! Who could blame you? It's a very reasonable thing to expect. Unfortunately for us the word 'language' has been hijacked by the formal-language people to mean something much more general than that. What they mean is the following.

We start with an **alphabet** Σ (and for some reason they always *are* called Σ, don't ask me why) which is a finite set of **characters**. We are interested in **string**s of characters from the alphabet in question. A *string* of characters is not the same a *set* of characters. Sets do not have order—the set $\{a, b\}$ is the same set as the set $\{b, a\}$ but strings do: the string ab is not the same as the string ba. Sets do not have *multiplicity*: the set $\{a, a, b\}$ is the same set as the set $\{a, b\}$ (which is why nobody writes things like '$\{a, a, b\}$') but strings definitely do have multiplicity: the string aab is not the same string as the string ab. Also, slightly confusingly, although we have this '{' and '}' notation for sets, there is no corresponding delimiter for strings. Notation was not designed rationally, but just evolved haphazardly.

We write 'Σ^*' for the set of all strings formed from characters in Σ, and then a **language** ("over Σ") is just a subset of Σ^*: any subset at all. A subset of Σ^* doesn't have to have a sensible definition in order to be called a language by the formal-language people. While we are about it, notice also that people use the word 'word' almost interchangeably with 'string'.

A word of warning. Many people confuse $\subseteq$ and $\in$, and there is a parallel confusion that occurs here. (This is not the same as the confusion between sets and strings: this is the common confusion between sets and their members!) A *language* is not a string: it is a *set* of strings. This may be because they are thinking that strings are sets of characters and that accordingly a language—on

this view—is a set of sets. If in addition you think that everything there is to be said about sets can be drawn in Venn diagrams, this will confuse you. Venn diagrams give you pictures of sets of *points*, but not sets of *sets*. A Venn diagram picture involving three levels of sets is impossible.

We will write '$|w|$' for the length of the string w. This may remind you of the notation '$|A|$' for the number of elements of A when A is a set.

(Aside on notation: the use of the asterisk in this way to mean something like "finite repetitions of" is widespread. If you are doing 116 you might connect this with the notation 'R^*' for the transitive closure of R. We will also see the notation 'σ^*' for the function that takes a state s, and a string w and returns the state that the machine reaches if we start it in s and then perform all the transitions mandated by the characters in w. Thus, for example, for characters a, b and c, $\sigma^*(s, ab) = \sigma(\sigma(s, a), b)$ and $\sigma^*(s, abc) = \sigma(\sigma(\sigma(s, a), b), c)$ and so on. In fact $\sigma^* : S \times \Sigma^* \to S$. It's easier than it looks!

Although a language is any old subset of Σ^*, on the whole we are interested only in languages that are infinite. And here I find that students need to be tipped off about the need to be careful. Print out this warning and pin it to the wall:

> **The languages we are interested in are usually infinite, but the strings in them are always finite!**

Let's have some examples. Let Σ be the alphabet $\{a, b\}$.

The language $\{aa, ab, ba, bb\}$ is the language of two-letter words over Σ. $\{a^n : n \in \mathbb{N}\}$ is the set of all strings consisting entirely of a's which is the (yes, it's infinite) language $\{\epsilon, a, aa, aaa, \ldots\}$. $\{a^n b^n : n \in \mathbb{N}\}$ is the set of all strings that consist of a's followed by the same number of b's.

Question: What might the symbol 'ϵ' mean? `miniexercise for JABBER session`

Mathematics is full of informal conventions that people respect because they find them helpful. Some of them apply here, and we wil respect them

1. We tend to use letters from near the beginning of the alphabet, a, b, ... for characters in alphabets;
2. We tend to use lower-case letters from near the end of the alphabet—like 'u', 'v' and 'w'—for variables to range over strings;
3. q is often a variable used to vary over states;
4. We tend to use upper-case letters from the middle of the alphabet—like 'K', 'L'—for variables to range over languages.

If w and u are strings then wu is w with u stuck on the end, and uw is u with w stuck on the end. Naturally $|uw| = |wu| = |w| + |u|$. ϵ is just the empty string (which we met on page 7 so ϵw—the empty string with w concatenated on the end—is just w. So ww, www and $wwwww$ are respectively the result of stringing two, three and five copies of w together. Instead of writing '$wwwww$' we write 'w^5' and we even use this notation for variable exponents, so that w^n is n copies of w strung together.

1.1.1 Languages from machines

Some languages have sensible definitions in terms of machines. There is an easy and natural way of associating languages with machines. It goes like this.

For each machine we single out one state as the designated "start" state.

Then we decorate some of the states of our machines with labels saying "accepting". My (highly personal and nonstandard!) notation for this is a smiley smacked on top of the circle corresponding to the state in question. The more usual notation is much less evocative: states are represented as circles, and accepting states are represented by a pair of concentric circles.

Aside on notation

> Some people write pictures of machines from which arrows might be missing, in that there could be a state s and a character c such that there is no arrow from s labelled c. With some people this means that you stay in s, and with some it means that you go from s to a terminally unhappy state—which I notate with a scowlie. My policy is to put in all arrows. This is a minority taste, but I think it makes things clearer. It's important to remember that what this affords us is not two examples of a different concept of machine, but different notations.

(For some reason the expression 'final state' is sometimes used for 'accepting state'. I don't like this notation, since it suggests that once you get the machine into that state it won't go any further or has to be reset or something, and this is not true. But you will see this nomenclature in the literature.)

The use of the word "accepting" is a give-away for the use we will put this labelling to. We say that a machine **accepts** a string if, when the machine is powered up in the designated start state, and is fed the characters from s in order, when it has read the last character of s it is in an accepting state.

This gives us a natural way of making machines correspond to languages. When one is shown a machine $\mathcal{M}$ one's thoughts naturally turn to the set of strings that $\mathcal{M}$ can accept—which is of course a language. We say that this set is the language **recognised by** $\mathcal{M}$, and we write it $L(\mathcal{M})$.

Pin this to the wall too:

> The set of strings accepted by a machine $\mathcal{M}$ is the language recognised by $\mathcal{M}$

People often confuse a machine-recognising-a-language with a machine-accepting-a-string. It is sort-of OK to *end up* confusing 'recognise' and 'accept' once you really know what's going on: lots of people do. However if you *start off* confusing them you will become hopelessly lost.

Two points to notice here. It's obvious that the language recognised by a machine is uniquely determined. However, if you start from the other direction, with a language and ask what machine determines it then it's not clear that

there will be a unique machine that recognises it. There may be lots or there may be none at all. The first possibility isn't one that will detain us long, but the second possibility will, for the difference between languages that have machines that accept them and languages that don't is a very important one. We even have a special word for them.

Definition:

> If there is a finite state machine which recognises L then L is **regular**.

Easy exercise. For any alphabet Σ whatever, the language Σ^* is regular. Exhibit a machine that recognises Σ^*. This is so easy you will probably suspect a trick.

`miniexercise for JABBER session`

1.1.2 Some exercises

1. (a) Draw a machine that recognises the language $\{\epsilon\}$. (This is easy but you might have to think hard about the notation!)

   (b) Draw a machine that recognises $\emptyset$, the empty language.

   `miniexercise for JABBER session`

2. Explain what languages the following notations represent

   (a) $\{a^{2n} : n \in \mathbb{N}\}$;

   (b) $\{(ab)^n : n \in \mathbb{N}\}$;

   (c) $\{a^n b^m : n < m \in \mathbb{N}\}$

   You might like to design machines to recognise the first two.

   `miniexercise for JABBER session`

3. For those of you who remember from 122 what CNF and DNF are. (If you don't yet know what CNF and DNF are, either find out or skip the question)

   (a) Propositional letters are p, p', p'', p''' .... This is a regular language.
   Exercise: Write a machine that recognises it.
   (Think: what is the alphabet?)

   (b) A literal is either a propositional letter, or a propositional letter preceded by a '$\neg$'.
   The set of literals forms a regular language.
   Exercise: Write a machine that recognises it.
   (Think: what is the alphabet?)

(c) A basic disjunction is a string like $p \vee \neg q \vee r$, namely a string of literals separated by '$\vee$'. (For the sake of simplicity we overlook the fact that no literal may occur twice!)

The set of basic disjunctions forms a regular language.

Exercise: Write a machine that recognises it.

(Think: what is the alphabet?)

(d) A formula in CNF is a string of basic disjunctions separated by $\wedge$, in the way that a basic disjunction is a set of literals separated by '$\vee$'.

The set of formulæ in CNF forms a regular language.

Exercise: Write a machine that recognises it.

(Think: what is the alphabet?)

1.2 The Thought-experiment

We live in a finite world, and all our machines are finite, so regular languages are very important to us, since they are the languages that are recognised by finite state machines. It's important to be able to spot (I nearly wrote 'recognise' there!) when a language is regular and when it isn't. Here is a thought-experiment that can help.

I am in a darkened room, whose sole feature of interest (since it has neither drinks cabinet nor coffee-making facilities) is a wee hatch through which somebody every now and then throws at me a character from the alphabet Σ. My only task is to say "yes" if the string of characters that I have had thrown at me so far is a member of L and "no" if it isn't (and these answers have to be correct!)

After a while the lack of coffee and a drinks cabinet becomes a bit much for me so I request a drinks break. At this point I need an understudy, and it is going to be you. Your task is to take over where I left off: that is, to continue to answer correctly "yes" or "no" depending on whether or not the string of characters that *we* (first I and then you) have been monitoring all morning is a member of L.

> **What are the parameters whose values I need to track? How many values can each parameter take? How much space do I require in order to store those values? What information do you want me to hand on to you when I go off for my drinks break? Can we devise in advance a form that I fill in and hand on to you when I go off duty?**

Let us try some examples

1. Let L be the set of strings over the alphabet $\{a, b\}$ which have the same number of as as bs. What do you want me to tell you?

   `miniexercise for JABBER session`

2. Let L be the set of strings over the alphabet $\{a, b\}$ which have an even number of as and an even number of bs. What do you want me to tell you?

   `miniexercise for JABBER session`

3. Let L be the set of strings over the alphabet $\{a, b\}$ where the number of as is divisible by 3 and so is number of bs. What do you want me to tell you?

   `miniexercise for JABBER session`

4. $\{a^n b^n : n \in \mathbb{N}\}$ This is the language of strings consisting of any number of as followed by the same number of bs. ('n' is a variable!)

   `miniexercise for JABBER session`

5. The "matching-brackets language": the set of strings of left and right brackets '(' and ')' that "match". (You know what I mean!)

   `miniexercise for JABBER session`

Each language L generates an equivalence relation on strings, and the thought-experiment is a way of getting us to think about it. While I am running the thought experiment I retain—each time a new character comes through the hatch to give me a string s—only that information which I will need in order to answer subsequent questions about whether or not later extensions of s are members of L. For example, when L is the language that consists of strings with an even number of as and an even number of bs, I don't bother distinguishing between—for example— *aabb* and *abab*. It's true they are different strings. But as far as I am concerned they are equivalent. Indeed, it's even true that for any string w of characters that I might later receive and put on the end of *aabb* or *abab*, *ababw* and *aabbw* are equivalent—in the same way *ababa* and *aabb* are.

As you remember from Discrete Maths 116 (And if you don't, well!—you know now!) to each equivalence relation there corresponds a set of equivalence classes. The key idea behind the thought-experiment is that the machine that you are imagining yourself to be is that machine whose states are equivalence classes of strings under this relation of equivalence-from-the-point-of-view-of-L. Do not worry if you do not see how to give a satisfactory mathematical-looking definition of this relation of equivalence-from-the-point-of-view-of-L: it's quite hard to do it properly, even though the idea should be appealing enough. The language L is regular if and only if the binary relation on strings of equivalence-from-the-point-of-view-of-L has only finitely many equivalence classes.

Equivalence-from-the-point-of-view-of-L must be a congruence relation for each of the $|\Sigma|$ operations "stick w on the end" (one for each $w \in \Sigma$). We saw these on page 6. This will all be in the new 116 notes.

1.3 The Pumping Lemma

The pumping lemma starts off as a straightforward application of the pigeonhole principle. If a machine $\mathcal{M}$ has n states, and accepts even one string that is of length greater than n, then in the course of reading (and ultimately accepting) that string it must have visited one of its states—s, say—twice. (At least twice: quite possibly more often even than that). This means that if w is a string accepted by $\mathcal{M}$, and its length is greater than n, then there is a decomposition of w as a concatenation of three strings $w_1w_2w_3$ where w_1 is the string of characters that takes it from the start state to the state s; w_2 is a string that takes it from s on a round trip back to s; and w_3 is a string that takes it from s on to an accepting state.

This in turn tells us that $\mathcal{M}$—having accepted $w_1w_2w_3$—must also accept $w_1(w_2)^nw_3$ for any n. $\mathcal{M}$ is a finite state machine and although it "knows" at any one moment which state it is *in at that moment* it has no recollection of its history, no recollection of how it got into that state nor of how often it has been in that state before.

Thus we have proved

> **The Pumping Lemma**
>
> If a finite state machine $\mathcal{M}$ has n states, and w is a string of length $> n$ that is accepted by $\mathcal{M}$, then there is a decomposition of w as a concatenation of three strings $w_1w_2w_3$ such that $\mathcal{M}$ also accepts all strings of the form $w_1(w_2)^nw_3$ for any n.

It does *not* imply that if w is a string of length $> n$ that is accepted by $\mathcal{M}$, and $w_1w_2w_3$ is *any old* decomposition of w as a concatenation of three strings then $\mathcal{M}$ also accepts all strings of the form $w_1(w_2)^nw_3$ for any n: it says merely that *there is at least one* such decomposition.

The pumping lemma is very useful for proving that languages aren't regular.

In order to determine whether a language is regular or not you need to form a hunch and back it. Either guess that it is regular and then find a machine that recognises it or form the hunch that it isn't and then use the pumping lemma. How do you form the hunch? Use the thought experiment. If the thought experiment tells you: "a finite amount of information" you immediately know it's a finite-state machine, and if you think about it, it becomes clear what the machine is. What do we do if the thought-experiment tells you that you need infinitely many states (beco's there appears to be no bound on the amount of information you might need to maintain)? This is where the pumping lemma comes into play. You use it to build *bombs*.

Bombs?! Read on.

1.4 Bombs

Let L be a language, and suppose that although L is not regular, there is nevertheless someone who claims to have a machine $\mathcal{M}$ that recognises it: i.e.,

they claim to have a machine that accepts members of L *and nothing else*. This person is the **Spiv**. This machine is fraudulent of course, but how do we prove it? What we need is a *bomb*.

A **bomb** (for $\mathcal{M}$) is a string that is either (i) a member of L not accepted by $\mathcal{M}$ or (more usually) (ii) a string accepted by $\mathcal{M}$ that isn't in L. Either way, it is a certificate of fraudulence of the machine $\mathcal{M}$, and therefore something that **explodes** those fraudulent claims.

How do we find bombs? This is where the pumping lemma comes in handy. The key to designing a bomb is feeding $\mathcal{M}$ a string w from L whose length is greater than the number of states of $\mathcal{M}$. $\mathcal{M}$ accepts w. $\mathcal{M}$ must have gone through a loop in so doing. Now we ascertain what substring w' of w sent $\mathcal{M}$ through the loop, and we insert lots of extra copies of that substring next to the one copy already there *and we know that the new "pumped" string will also be accepted by* $\mathcal{M}$. With any luck it won't be in L, and so it will be a bomb. The key idea here is that *the machine has no memory of what has happened to it beyond what is encoded by it being in one state rather than another.* So it cannot tell how often it has been though a loop. *We*—the bystanders—know how often it has been sent through a loop but the machine itself has no idea.

Examples are always a help, so let us consider some actual challenges to the bomb-maker.

1.4.1 One-step refutations using bombs

The language $\{a^n b^n : n \in \mathbb{N}\}$ is not regular

Suppose the Spiv shows us $\mathcal{M}$, a finite state machine that—according to him—recognises the language $\{a^n b^n : n \geq 0\}$. The thought-experiment tells us immediately that this language is not regular (this was example 4 on page 11) so we embark on the search for a bomb with high expectations of success. In the first instance the only information we require of the Spiv is the number of states of $\mathcal{M}$, though we will be back later for some information about the state diagram. For the moment let k be any number such that $2k$ is larger than the number of states of $\mathcal{M}$. Think about the string $a^k b^k$. What does $\mathcal{M}$ do when fed $a^k b^k$? It must go through a loop of course, because we have made k large enough to ensure that it does. In the process of going through the loop it is reading a substring w of $a^k b^k$. What does the substring consist of? We don't know the exact answer to this, but we can narrow it down to one of the three following possibilities, and in each case we can design a bomb.

1. w consists entirely of as. Then we can take our bomb to be the string obtained from $a^k b^k$ by putting n copies of w instead of just one. Using our notation to the full, we can notate this string $a^{(k-|w|)}a^{(|w|\cdot n)}b^k$

   Explain why this is correct. `miniexercise for JABBER session`

2. w consists entirely of bs. Then our bomb will be the string obtained from $a^k b^k$ by putting in several copies of w instead of just one. $\mathcal{M}$ will accept this string, but this string contains more bs than as, and $\mathcal{M}$ shouldn't

have accepted it. Exercise: how do we notate this bomb, by analogy with the bomb in the previous case? `miniexercise for JABBER session`

3. w consists of some as followed by some bs. In this case, when we insert n copies of w to obtain our bomb, we compel $\mathcal{M}$ to accept a string that contains some as followed by some bs and then some as again (and then some bs). But—by saying that $\mathcal{M}$ recognised $\{a^n b^n : n \in \mathbb{N}\}$ the Spiv implicitly assured us that the machine would not accept any string containing as *after* bs. Exercise: how do we notate this bomb, by analogy with the bomb in the previous case? `miniexercise for JABBER session`

So we know we are going to be able to make a bomb whatever w is. However, if we are required to actually exhibit a bomb we will have to require the Spiv to tell us what w is.

The language $\{a^k b a^k : k \geq 0\}$ is not regular

Suppose the Spiv turns up with $\mathcal{M}$, a finite state machine that is alleged to recognise the language $\{a^k b a^k : k \geq 0\}$. Notice that every string in this language is a palindrome (a string that is the same read backwards as read forwwards). We will show that $\mathcal{M}$ will accept some strings that aren't palindromes, and therefore doesn't recognise $\{a^k b a^k : k \geq 0\}$.

As before, we ask the Spiv for the number of states the machine has, and get the answer m, say. Let n be any number bigger than m and consider what happens when we give $\mathcal{M}$ the string $a^n b a^n$. This will send $\mathcal{M}$ through a loop. That is to say, this string $a^n b a^n$ has a substring within it which corresponds to the machine's passage through the loop. Call this string w. Now, since we have force-fed the machine n a's, and it has fewer than n states, it follows that w consists entirely of a's. Now we take our string $a^n b a^n$ and modify it by replacing the substring w by lots of copies of itself. Any number of copies will do, as long as it's more than one. This modified string (namely $a^{(k+|w|)} b a^k$) is our bomb. Thus $\mathcal{M}$ accepts our bomb, thereby demonstrating—as desired—that it doesn't recognise $\{a^k b a^k : k \geq 0\}$.

1.4.2 A few more corollaries

1. $\mathcal{M}$, if it accepts anything at all, will accept a string of length less than $|\mathcal{M}|$.

2. If $\mathcal{M}$, accepts even one string that has more characters than $\mathcal{M}$ has states will accept arbitrarily long strings.

The Pumping Lemma is a wonderful illustration of the power of **The Pigeonhole Principle**. If you have $n + 1$ pigeons and only n pigeonholes to put them in the at least one pigeonhole will have more than one pigeon in it. The pigeonhole principle sounds too obvious to be worth noting, but the pumping lemma shows that it is very fertile.

Chapter 2

Operations on machines and languages

Languages are sets, and there are operations one can perform on them simply in virtue of their being sets. If K and L are languages, so obviously are $K \cup L$, $K \cap L$ and $K \setminus L$. There is one further operation that we need which is defined only because languages are sets of strings and there are operations we can perform on strings, specifically concatenation. This concatenation of strings gives us a notion of concatenation of languages. $KL = \{wu : w \in K \wedge u \in L\}$, and the reader can probably guess what K^* is going to be. It's the union of K, KK, KKK ...

Exercise: If $K = \{aa, ab, bc\}$ and $L = \{bb, ac, ab\}$ what are (i) KL, (ii) LK, (iii) KK, (iv) LL? `miniexercise for JABBER session`

Exercise: Can you express $|KL|$ in terms of $|K|$ and $|L|$?

`miniexercise for JABBER session`

The following fact is fundamental.

> If K and L are regular languages so are $K \cap L$, $K \setminus L$, KL and K^*.

So every language you can obtain from regular languages by means of any of the operations we have just seen is likewise regular.

Let's talk through this result.

$K \cap L$

The thought-experiment shows very clearly that the intersection of two regular languages is regular : if I have a machine $\mathcal{M}_1$ that recognises L_1 and a machine $\mathcal{M}_2$ that recognises L_2 I obviously run them in parallel, giving each new incoming character to both machines and accept a string if they both accept it. Using the imagery of the thought-experiment, the clipboard of information that I hand on to you when I go for my coffee break has become a pair of clipboards, one for $\mathcal{M}_1$ and one for $\mathcal{M}_2$.

But although this makes it clear that the intersection of two regular languages is regular, it doesn't make clear what the machine is that recognises the intersection. We want to cook up a machine $\mathcal{M}_3$ such that we can see running-$\mathcal{M}_1$-and-$\mathcal{M}_2$-in-parallel as merely running $\mathcal{M}_3$. $\mathcal{M}_3$ is a sort of composite of $\mathcal{M}_1$ and $\mathcal{M}_2$. What are the states of this new composite machine?

The way to see this is to recall from page 11 the idea that a state of a machine is a state-of-knowledge about the string-seen-so-far. What state-of-knowledge is encoded by a state of the composite machine? Obviously a state of the composite machine must encode your knowledge of the states of the two machines that have been composed to make the new (composite) machine. So states of the composite machine are ordered pairs of states of $\mathcal{M}_1$ and $\mathcal{M}_2$.

What is the state transition function for the new machine? Suppose $\mathcal{M}_1$ has a transition function σ_1 and $\mathcal{M}_2$ has a transition function σ_2, then the new machine has the transition function that takes the new state $\langle s_1, s_2 \rangle$ and a character c and returns the new state $\langle \sigma_1(s_1, c), \sigma_2(s_2, c) \rangle$.

$K \cap L$ and $K \setminus L$

The proofs that a union or difference of two regular languages is regular is precisely analogous.

People sometimes talk of the *complement* of a language L. L is a language over an alphabet Σ, and its complement, relative to Σ, is $\Sigma^* \setminus L$. It's easy to see that the complement of a regular language L is regular: if we have a machine $\mathcal{M}$ that recognises L, then we can obtain from it a machine that recognises the complement of L just by turning all accepting states into non-accepting states and *vice versa*.

(A common mistake is to assume that every subset of a regular language is regular. I think there must be a temptation to assume that a subset of a language recognised by $\mathcal{M}$ will be recognised by a version of $\mathcal{M}$ obtained by throwing away some accepting states.)

KL

It's not obvious that the concatenation of two regular languages is regular, but it's plausible. We will explain this later. For the moment we will take it as read and press ahead. This leads us to

2.1 Regular Expressions

A regular expression is a formula of a special kind. Regular expressions provide a notation for regular languages. We can declare them in BNF (Backus-Naur form).

We define the class of regular expressions over an alphabet Σ recursively as follows

1. An element of Σ is a regular expression;

2. If A is a regular expressions, so is A^*;

3. If A and B are regular expressions, so is AB;

4. If A and B are regular expressions, so is $A|B$.

For example, for any character a, a^* is a regular expression.

The idea then is that regular expressions built up in this way from characters in an alphabet Σ will somehow point to regular languages $\subseteq \Sigma^*$. Now we are going to recall the $L()$ notation which we used on page 8, where $L(\mathcal{M})$ is the language recognised by $\mathcal{M}$, and overload it to notate a way of getting languages from regular expressions. We do this by recursion using the clauses above.

1. If a is a character from Σ (and thus by clause 1, a regular expression) then $L(a) = \{a\}$.

2. $L(A|B) = L(A) \cup L(B)$.

3. $L(AB) = L(A)L(B)$. This looks a bit cryptic, but actually makes perfect sense. $L(A)$ and $L(B)$ are languages. If K_1 and K_2 are languages, you know what K_1K_2 is from page 15, so you know what $L(A)L(B)$ is!

   $L(A^*)$ is the set $L(A) \cup L(AA) \cup L(AAA) \ldots = \bigcup_{n \in \mathbb{N}} A^n$. A^n is naturally $AAAA \ldots A$ (n times).

2.1.1 More about bombs

2.1.1.1 Palindromes do not form a regular language

You may recall that a palindrome is a string that it the same read backwards or forwards. If you ignore the spaces and the punctuation then the strings 'Madam, I'm Adam' and 'A man, a plan, a canal—Panama!" are palindromes.

(Even better: A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a peon, a canal—Panama!!)

The thought-experiment swiftly persuades us that the set of palindromes over an alphabet Σ is not regular (unless Σ contains only one character of course!). After all—as you will have found by looking first at "Madam, I'm Adam" and then the two longer examples—to check whether or not a string is a palindrome one finds oneself making several passes through it, and having to comapre things that are arbitrarily far apart.

Let L be the language of palindromes over $\{a, b\}$. It isn't regular, but there is no obvious bomb. However, if L were regular then so too would be the language $L \cap L(a^*ba^*)$. (We established on page 15 that the intersection of two regular languages is regular.) This new language is just the language $\{a^kba^k : n \in \mathbb{N}\}$ that we saw on page 14.

2.1.1.2 The language $\{ww : w \in \Sigma^*\}$ is not regular

I don't see how to use a bomb to show that $\{ww : w \in \Sigma^*\}$ is not regular, though it's obvious from the thought-experiment. However, we do know that the language $L(a^*b^*a^*b^*)$ is regular so if our candidate were regular so too would be the language $\{ww : w \in \Sigma^*\} \cap L(a^*b^*a^*b^*)$. Now this language is $\{a^n b^m a^n b^m : m, n \in \mathbb{N}\}$ and it isn't hard to find bombs to explode machines purporting to recognise this language. You might like to complete the proof by finding such a bomb.

`miniexercise for JABBER session`

2.2 Kleene's theorem

this section is not examinable this time, and may be ommitted. However, any serious student who wiishes to master automata theory will want to read it. There is good coverage of Kleene's theorem in Professor Pitts' notes.

Kleene's theorem states that a language is regular if it can be notated by a regular expression. One direction of this is fairly easy: showing that if there is a regular expression for a language L then there is a machine that recognises L. This breaks down into several steps, one for each constructor: slash, concatenation and Kleene star. We've seen how to do slash—after all $L(K_1|K_2)$ is just $L(K_1) \cup L(K_2)$—but to do the other two involves nondeterministic machines and we don't encounter those until later.

The hard part is the other direction: showing how to find a regular expression for the language recognised by a given machine.

What we prove is something apparently much stronger:

For every machine $\mathcal{M}$, for any two states q_1 and q_2 of $\mathcal{M}$, and for any set Q of states, there is a regular expression $\phi(q_1, q_2, Q)$ which notates the set of strings that take us from state q_1 to state q_2 while never leaving the set Q.

Of course all we are after is the regular expression formed by putting slashes between all the expression $\phi(q_1, q_2, Q)$ where q_1 is the initial state, q_2 is an accepting state, and Q is the set of all states. But it turns out that the only way to prove this special case is to prove the much more general assertion.

We prove this general assertion by induction. The only way to have a hope of understanding this proof is to be quite clear about what it is we are proving by induction. You are probably accustomed to having 'n' as the induction variable in your proofs by induction so let's do that here.

"For all machines $\mathcal{M}$, and for all subsets Q of the set of $\mathcal{M}$'s states with $|Q| = n$, and for any two states q_1 and q_2 of $\mathcal{M}$, there is a regular expression $\phi(q_1, q_2, Q)$ which notates the set of strings that take us from state q_1 to state q_2 while never leaving the set Q"

We fix $\mathcal{M}$ once for all (so that we are doing a '$\forall$-introduction rule, or "universal generalisation" on the variable '$\mathcal{M}$', and we prove by induction on 'n' that this is true for all n.

At the risk of tempting fate, I am inclined to say at this point that if you

are happy with what has gone so far in this section (and that is quite a big if!) then you have done all the hard work. The proof by induction is not very hard. The hard part lay in seeing that you had to prove the more general assertion first and then derive Kleene's theorem as a consequence.

Proofs by induction all have two parts. (i) A base case, and (ii) induction step. I submit, ladies and gentlemen, that the base case—with $n = 1$—is obvious. Whatever $\mathcal{M}$, s and q are, you can either get from q_1 to q_2 in one hop by means of—character c, say (in which case the regular expression is c)—or you can't, in which case the regular expression is ϵ.

Now let's think about the induction step. Suppose our assertion true for n. We want to prove it true for $n + 1$.

We are given a machine $\mathcal{M}$, and two states q_1 and q_2 of $\mathcal{M}$. We want to show that for any set Q of states of $\mathcal{M}$, with $|Q| = n + 1$, there is a regular expression that captures the strings that take the machine from q_1 to q_2 without leaving Q.

What are we allowed to assume? The induction hypothesis tells us that for any two states s' and t' and any set Q' of states with $|Q'| = n$, there is a regular expression that captures the strings that take the machine from q_1' to q_2' without leaving Q'. (I have written 'q_1'' and 'q_2'' and Q' because I don't want to reuse the same variables!)

For any state r in Q, we can reason as follows: "Every string that takes $\mathcal{M}$ from q_1 to q_2 without leaving Q either goes through r or it doesn't.

The strings that take $\mathcal{M}$ from q_1 to q_2 without either going through r or leaving Q are captured by a regular expression because $|Q \setminus \{r\}| = n$. Let w_1 be this regular expression.

The strings that take $\mathcal{M}$ from q_1 to q_2 *via* r are slightly more complicated. By induction hypothesis we have a regular expression for the set of strings that take $\mathcal{M}$ from q_1 to r without going through q_2 (while remaining in Q)—because $|Q \setminus \{q_2\}| = n$—so let's call that regular expression w_2. Similarly by induction hypothesis we have a regular expression for the set of strings that take $\mathcal{M}$ from r to q_2 without going through q_1 (while remaining in Q)—because $|Q\setminus\{q_1\}| = n$—so let's call that regular expression w_3. Finally by induction hypothesis we have a regular expression for the set of strings that take $\mathcal{M}$ from r back to r without going through q_1 or q_2 (while remaining in Q)—because $|Q\setminus\{q_1, q_2\}| = n-1$—so let's call that regular expression w_4.

Now a string that takes $\mathcal{M}$ from q_1 to q_2 *via* r will consist of a segment that takes $\mathcal{M}$ from q_1 to r (captured by w_2) followed by a bit that takes it from r back to r any number of times (captured by w_4^*) followed by a bit that takes $\mathcal{M}$ from r to q_2 (captured by w_3).

So the regular expression we want is $w_1|w_2(w_4)^*w_3$.

This concludes the proof

If you are a confident and fluent programmer in a language that handles strings naturally then you should try to program the algorithm on which this proof relies. It will give you good exercise in programming and will help you understand the algorithm.

[*HOLE*]

2.2.1 Some more exercises

1. Is every finite language regular?
2. The "reverse" of a regular language is regular: if L is regular, so is $\{w^{-1} : w \in L\}$ where w^{-1} is w written backwards.
3. You know that you cannot build a finite state machine that recognises the matching bracket language. However you can build a machine that accepts all and only those strings of matching brackets where the number of outstanding opened left brackets never exceeds three.

   Find a regular expression for the language accepted by this machine. (I suggest that, in order not to drive yourself crazy, you write '0' instead of the left bracket and '1' instead of the right bracket!!

   `miniexercise for JABBER session`

   How about the same for a machine than can cope with as many as five outstanding brackets?

   `miniexercise for JABBER session`
4. Find regular expressions for the regular languages in the CNF exercise on page 9.

2.3 Arden's rule and some stuff like that

B I G H O L E H E R E

Chapter 3

Grammars

So far we have encountered two ways of thinking about regular languages: (i) through finite state machines; (ii) through regular expressions. These approaches have their roots in the study of machines, rather than—as you had probably been expecting—the study of natural languages. The third approach, you will be relieved to hear, is one that has its roots in the study of natural languages after all.

Many years ago (when I was at school) children were taught *parsing*. We were told things like the following. Sentences break down into **subject** followed by **verb** followed by **object**. Or perhaps they break down into **Noun Phrase** followed by **verb** followed by **Noun Phrase**. These constituents break down in turn: noun phrases being composed of **determiner** followed by **adjective** followed by **noun**.

This breaking-down process is important. The idea is that the way in which we assemble a sentence from the parts into which we have broken it down will tell us how to recover the meaning of a sentence from the meanings of its constituents.

Rules like

sentence → Subject verb object

and

Noun phrase → article adjective noun

have the potential, once equipped with further rules like

verb → swim

verb → walk

noun → dog

noun → cat

to generate words in a language. This time 'language' really does mean something like 'language' in an ordinary sense. But the "words" that we generate are actually things that in an ordinary context woul dbe called *sentences*. And

this time we think of 'dog' not as a string but as a character from an alphabet, don't we!

The languages that we have seen earlier in this coursework can be generated by rules like this. For example

$$S \rightarrow aS$$

$$S \rightarrow \epsilon$$

generates every string in the language $L(a^*)$ and a set of rules like

$$S \rightarrow aSa$$

$$S \rightarrow bSb$$

$$S \rightarrow \epsilon$$

generates the language of palindromes over the alphabet $\Sigma = \{a, b\}$.

These bundles of rules are called **grammars** and the rules in each bundle are called **productions**.

There are two sorts of characters thatappear in productions. There are **nonterminals** which appear on the left of productions (and sometimes on the right). These are things like 'noun' and 'verb'. **Terminals** are the characters from the alphabet of the language we are trying to build, and they only every appear on the right-hand-side of the production. Examples here are 'swim', 'walk', 'cat' and 'dog'.

Notice that there is a grammar that generates the language of palindromes over $\Sigma = \{a, b\}$ even though this language is not regular. Grammars that generate regular languages have special features that mark them off. One can ascertain what these features are by reverse-engineering the definition of regular language from regular expressions or finite state machines, but we might as well just give it straight off.

A Regular grammar is one in which all productions are of the form

$$N \rightarrow TN'$$

or

$$N \rightarrow T$$

Where N and N' are nonterminals and T is a string of terminals.

The other illustrations i have given are of grammars not having this restriction. They are context-free. Reason for this nomenclature is burble...

3.0.1 Exercises

Provide a context-free grammar for regular expressions over the alphabet $\{a, b\}$

3.1 Pushdown Automata

We have characterised context-free languages in terms of grammars, but they can also be characterised in terms of machines. There is a special kind—a special class—of machines which we call **push-down automata** (or"PDA" for short) that are related to context-free languages in the same way that finite state machines are related to regular languages. Just as a language is regular iff there is a finite-state machine that recognises it (accepts all the strings in it and accepts no other strings), so a language is context-free iff there is a pushdown automaton that recognises it.

So what is a push-down automaton? One way in to this is to first reflect on what extra bells and whistles a finite state machine must be given if it is to recognise a context-free language such as the matching-brackets language, and to then engineer those bells and whistles into the machine. The thought-experiment comes in handy here. What does the thought-experiment tell you about the matching backets language? Run the experiment and you will find that to crack the matching-brackets language it would be really nice to have a *stack*. Every time you see a left bracket you push it onto the stack and every time you see a right-bracket you pop a left-bracket off the stack and whenever the stack is empty you are in an accepting state. If you ever find yourself trying to pop a bracket off an empty stack you know that things have gone permanently wrong so you shunt yourself into a scowlie. What I need to hand over to you when I go off for my coffee is the stack (or the scowlie).

Let us now try to formalise the idea of a machine-with a stack. Our way in is to start off by thinking of a PDA as a finite state machine which we are going to upgrade, so we have the idea of start state and accepting state as before. The stack of course contains a string of characters. For reasons of hygiene we will tend to assume that the alphabet of characters that go on the stack (the "pushdown alphabet") is disjoint from the alphabet that the context free language is drawn from. Clearly if there is to be any point in having a stack at all then the machine is going to have to read it, and this entails immediately that the transition function of the machine has not *two* arguments (as the transition function of a finite state machine does, namely the old state and the character being read) but *three*, with the novel third argument being the character at the top of the stack. And of course the transition function under this new arrangement not only tells you what state the machine will go into, but what to do with the stack—namely push onto it a word (possibly empty) from the stack alphabet.

(This is a bit confusing: the transition function in the new scheme of things takes an input which is an ordered triple of the contents-of-the-stack, the new character that is being fed it by the user, and the current state; the value is a pair of a state and the new contents-of-the-stack. However the transition function doesn't look at the whole of the contents of the stack but only the top element. Specifically this means that the new stack can differ from the old in only very limited ways. The new stack is always the old stack with the top element replaced by a string (possibly empty) of characters from the stack

alphabet.)

If you are happy with this description you might now like to try designing a PDA that accepts the matching bracket language. (Hint: it has only three states: (i) accepting, (ii) wait-and-see, and (iii) dead!)

`miniexercise for JABBER session`

There is a slight complication in that PDA's are nondeterministic, so the parallel is with regular languages being recognised by nondeterministic finite automata rather than FSAs... But we haven't dealt with nondeterministic machines yet. So we'd better deal with them at once!

3.2 Exercises

Write context-free grammars for conjunctive normal form and disjunctive normal form. See page 9.

`miniexercise for JABBER session`

Chapter 4

Nondeterministic Machines

A nondeterministic machine is just like a deterministic machine except that its transition behaviour isn't deterministic. If you know the state a deterministic machine $\mathcal{M}$ is in then you know what state it will go to when you give it character a (or b or whatever). With a nondeterministic machine you only know the set of states that it *might* go to next. Notice that a deterministic machine is simply a nondeterministic machine where this set-of-states-that-it-might-go-to is always a singleton.

Nondeterministic machines (hereafter **NFA**s—"nondeterministic finite automata") are a conceptual nightmare. The fact that they are nondeterministic makes for a crucial difference between them and deterministic machines. In the deterministic case you don't have to distinguish in your mind between its behaviour in principle and its behaviour in practice, since its behaviour in practice is perfectly reproducible. That means that you can think of a deterministic machine either as an abstract machine—a drawing perhaps—or as a physical machine, according to taste. With NFAs there is a much stronger temptation to think of them as actual physical devices whose behaviour is uncertain, rather than as abstract objects. And the difficulty then is that NFAs are not physically realisable in the way one would like.

So if NFAs are so nasty, why do we study them? The answer is that they tie up some loose ends and enable us to give a smooth theoretical treatment that improves our understanding and appreciation. So let us get straight what they are for. We started this course with a connection between machines and languages. A machine accepts strings and recognises a language. A (physical) nondeterministic machine can accept strings in exactly the same way a (physical) deterministic one does. You power it up, and feed in the characters one-by-one and when it's finished reading the string its either in an accepting state or it isn't. The subtlety is that a nondeterministic machine, on having read a string, might be in any of several states, perhaps some of which are accepting and perhaps some not. The only sensible definition we can give of an (abstract) nondeterministic machine recognising a language is this:

The language recognised by a nondeterministic machine $\mathcal{M}$ is the set of strings that one of its physical realisations might accept.

The task of remembering and understanding this definition is made much easier for you once you notice that the definition for recognition of languages by deterministic machines is simply a special case of this.

Nondeterministic machines are useful to us because of the combination of two facts.

(i) If L is a language recognised by a nondeterministic machine $\mathcal{M}$ then there is a deterministic machine $\mathcal{M}'$ which can be obtained in a systematic way from $\mathcal{M}$ that also recognises L.

(ii) There are circumstances in which it is very easy to produce a nondeterministic machine that recognises a language but no obvious easy way to produce a deterministic one.

Let us now prove (i) and illustrate (ii).

(i): Finding a DFA that emulates an NFA

(i) Suppose I have a nondeterministic machine $\mathcal{M}$, presented to me in its start state. I have a handful of characters $\{c_1, c_2, c_3 \ldots\}$ that I feed to the machine one by one. Initially I know the machine is in the start state. But after I've given it c_1 I know only that it is in one of the states that it can go to from the start state on being given c_1. And after I've given it c_2 I know only that it is one of those states it can reach from the start state in two hops if given c_1 followed by c_2 and so on. We seem to be losing information all the time. But all is not lost. Although I do not have certain knowledge of *the state $\mathcal{M}$ is in*, I do nevertheless have certain knowledge of *the set of states that it might be in*. And this is something I can keep track of, in the following sense. I can say "If it's in one of the states s or s' or s'' and I give it character c then either it was in s in which case it's now in s''' or s'''' or it was in s' in which case it's now in $\ldots$. In other words

1. if I know the set of states that it might be in now (and it must be in one of them) and
2. I know the character it is being given, then I know
3. the the set of states that it might be in next (and it must be in one of them).

Now comes the trick. Think of the set-of-states-that-it-might-be-in as a state of a new machine! One way of seeing this is to think of the states of the new deterministic machine as the states of uncertainty you might be in about the state of the nondeterministic machine. We have seen something like this

before: in the discussion of the thought-experiment we were viewing states of the machine as states of knowledge of the string-so-far; this time we are thinking of states of the new (deterministic) machine as states-of-knowledge-of-what-state-the-nondeterministic-machine-might-be-in.

(ii) An application of NFAs

I mentioned earlier that the concatenation of two regular languages is regular. Suppose I have a deterministic machine $\mathcal{M}_1$ that recognises L and a deterministic machine $\mathcal{M}_2$ that recognises K. The idea is to somehow "stick $\mathcal{M}_2$ on the end of $\mathcal{M}_1$".

The difficulty is that if w is a string in LK, it might be in LK for more than one reason, since it might be decomposible into a string-from-K followed by a string-from-L in more than one way. So one can't design a machine for recognising LK by saying "I'll look for a string in K and then—when I find one—swap to looking for a string in L". You have to start off imagining that you are in $\mathcal{M}_1$; that much is true. However when you reach an accepting state you have to choose between (i) staying in $\mathcal{M}_1$ and (ii) making an instantaneous hop through a trap-door to the start-state of $\mathcal{M}_2$. That is where the nondeterminism comes in. These instantaneous hops are called "ϵ-**transitions**". You do them *between* the clock ticks at which you receive new characters. I don't like ϵ-**transitions** and I prefer theoretical treatments that don't use them. However, they do appear in the literature and you may wish to read up about them.

For those who do like ϵ-transitions, here is a description of a nondeterministic machine that recognises LK. It looks like the disjoint union $\mathcal{M}_1 \sqcup \mathcal{M}_2$ of $\mathcal{M}_1$ and $\mathcal{M}_2$. Transitions between the states of $\mathcal{M}_1$ are as in $\mathcal{M}_1$ and transitions between the states of $\mathcal{M}_2$ are as in $\mathcal{M}_2$. In addition for each accepting state of $\mathcal{M}_1$ there is a ϵ-transition to the start state of $\mathcal{M}_2$.

disjoint unions from discrete structures

For those of you who—like me—do not like ϵ-transitions, here is a different nondeterministic machine that recognises LK. Like the last one, it looks like the disjoint union $\mathcal{M}_1 \sqcup \mathcal{M}_2$ of $\mathcal{M}_1$ and $\mathcal{M}_2$. Transitions between the states of $\mathcal{M}_1$ are as in $\mathcal{M}_1$ and transitions between the states of $\mathcal{M}_2$ are as in $\mathcal{M}_2$. In addition, whenever s is a state of $\mathcal{M}_1$ and c a character such that $\sigma(s, c)$ is an accepting state of $\mathcal{M}_1$, we put in an extra arrow from s to the start state of $\mathcal{M}_2$, and label this new arrow with a 'c' too. The effect of this is that when you are in s and you receive a c, you have to guess whether to stay in $\mathcal{M}_1$ (by going to an accepting state in $\mathcal{M}_1$) or make the career move of deciding that the future of the string lies with K, in which case you move to the start state of $\mathcal{M}_2$.

The manner in which we got rid of ϵ-transitions in this case is perfectly general. You can always get rid of them by introducing a bit of nondeterminism in the way we have just done.

And you can go in the other direction too.