Regular Languages and Finite Automata

A C Norman, Lent Term 1996

Part IA

1 Introduction

This course is short, but it is present in Part 1A because of the way it introduces links between many different parts of Computer Science and explains concepts that you will come across repeatedly throughout the rest of the course. One result of the topic being multi-faceted is that there are quite a few different ways of presenting it. Lectures covering this material were given by Andy Pitts in 1995 (to Part IA) and by Ken Moody to the Diploma in Computer Science class this year. Copies of their lecture notes may be available as backup to these ones, but although the major results they present are just the same as the ones I do the order in which they cover things and the way in which they prove results will generally differ from mine. I view this as good, in that it means that strong students have an opportunity to see two or three different perspectives on the material, while those who find my explanations hard going may find those in one of the other sets of notes more comfortable. When it comes to examination questions and the like it will not matter whose or which proofs of theorems you use, provided you get the details correct in the proofs that you do use.

I decided that I would recommend just one book: "Languages and Machines" by Sudkamp[1]. There will be plenty of other textbooks that cover the material in perfectly adequate detail. Since this is only a six lecture course the important parts of it will tend to form just one or two chapters in a book: In Sudkamp it is chapters 6 and 7 that are most relevant.

These notes are **not** a complete substitute for the textbook. In particular the book contains lots of examples and exercises, while these notes do not. Also the document preparation system I am using to prepare these notes means that it is a bit of a pain for me to prepare pictures, and as you will see on the blackboard during lectures these are a useful way of giving an informal illustration of a finite machine—the book is better illustrated than these notes. Once again I will stress that if you are going to understand this course fully you will need to work through collections of little examples. As well as finding some in the book you can look in past examination papers (until recently this topic was in Part IB, and it has been and still is presented to the Diploma students, so those are the papers to check).

2 What is this course about?

The main result presented here is that several areas that relate to computers and that start off looking as if they are quite separate are in fact very closely related. The word used to indicate the commonality is "regular".

2.1 Hardware made using flip-flops

In the Part 1A Digital Electronics course you learn how to design circuits using flip-flops. In general the circuits involved will have several inputs (in addition to a clock) and several outputs. You may consider the difference between Mealy Machines (where the output depends on both input and state) or Moore machines (where the output depends only on the state of the flip-flops). In a hardware course it is also necessary to think about the difference between (say) J-K and D flip-flops, and to wonder how to build circuits using the smallest possible number of components. In this course we look at a clean abstraction of all this. A "Finite Automaton"[1] will be a system with a finite number of distinct internal states. Rather than having several separate input wires we view it as being offered input *symbols* from some *alphabet*. Each symbol of course encodes a possible configuration of signals on the input wires of any physical realisation of the automaton. One nominated state of the machine is its *initial state*, and each time it is presented with an input symbol it moves to a new state in accordance with some definite rule. This rule can be viewed as an *transition function* where the new state is selected on the basis of the previous state and the symbol seen. To keep things as simple as possible, the automaton only has one (binary) output, and that depends solely on which state it is in (ie we have a Moore machine). This output arrangement can be described by listing each of the states where the output is **true**. Such states are then referred to as *accepting states*. The way such systems will be looked at will involve considering what sequences of input symbols cause the machine to enter an yield an output **true**: such input sequences are said to be *accepted* by the machine.

Looking ahead to where we need to conduct some proofs, I will formalise the above by suggesting that a finite automaton can be represented using the following five components:

A set of states: Q. I will generally write the names of individual states as letters with subscripts, so say $Q = \{q_i | 1 \leq i \leq N\}$ for some natural number N;

An alphabet: Σ. In my examples I will use lower case letters as "symbols", so for instance $\Sigma = \{a, b, c\}$. Note that I will insist on having a definite finite alphabet associated with any machine;

An initial state: By convention I will use q_0;

A transition function: A (total) function from $Q \times \Sigma$ to Q;

A set of accepting states: A subset of Q.

[1] Also sometimes known as Finite Machines, or Finite State Machines, and later in this course as Finite Deterministic Acceptors, FDAs!

This course will provide ways of describing exactly what automata of this sort can do. A useful idea you might like to hold in your mind is of automata that are to be used as combination locks. The lock starts off in some defined initial state, and the user feeds it symbols. In some configurations the lock "accepts" the sequence of symbols seen thus far and presumably allows the user to open the door to the bank vaults.

Viewing this course as one that talks about possible behaviours of sequential hardware gives the course a strong link with other hardware courses.

2.2 Discrete Mathematics—and beyond

You have just seen that I set up a description as a slightly idealised bit of hardware as a collection of five items each of which looks just as if it came out of the Discrete Mathematics course. This is not an accident! In this course I am able to develop part of a mathematically styled description of hardware behaviour. The value of using mathematical notation is that it helps us when we need to produce formal proofs. We need these proofs because some of the properties of finite hardware and the other things I describe are not intuitively obvious. The main results that will be proved here are ones that show that certain forms of behaviour can be realised using hardware build out of finite collections of flip flops and gates and that some others can not. I will also be able to prove that (at least in principle) it is always possible to tell if two circuits implement the same behaviour. These proofs will use some of the mathematical notation and techniques from the Discrete Mathematics course and represent an early clear-cut Computer Science pay-off from it.

In some sense **any** real physical computer is finite and so falls within the scope of the analysis done in this course. However it is generally much more useful to agree that the processing and arithmetic parts of a computer are finite (and hence belong here) but to imagine that the discs and memory attached are of unlimited size. That idealisation is considered in some depth in a Part IB course (Computation Theory), which to a large extent can be thought of as a follow on to this one. It is clearly important to understand the capabilities and limitations of finite automata before worrying about ones that have unbounded memory at their disposal! The The Computation Theory course is again able to develop a mathematically styled analysis of the behaviours of the computing systems that it considers.

2.3 Formal description of "languages"

Now, and perhaps amazingly, I will show that there is a link between this course and the Part IA courses that involve programming, ie the ones that use ML and Modula-3. The link comes about because this course can be seen as a precursor to

Part IB ones on how compilers work. Part of the process of designing a computer language involves setting up a precise description of what sequences of symbols and words can form valid fragments of program. Part of writing the software that implements a programming language involves seeing what the user had written and deciding if it matches what the language designer wanted. Some things called *phrase structure grammars* provide a well established tool for describing languages. These grammars work using an alphabet of *terminal* symbols and a set of *non-terminals*. The understanding is that terminal symbols are items that the user sees or puts into the text of a program, while non-terminals (for which I will use upper case here) control the internal workings of the grammar. A particular grammar will always have one nominated non-terminal that is its *initial symbol* (I will generally use a symbol called S for this), and a set[2] of *production rules*. For the purposes of **this** course I will consider production rules of somewhat limited forms. The limitations involved lead to what are known as *regular* grammars. They are the simplest sorts of phrase structure grammars, and an understanding of them is clearly useful before moving on to more general cases. In these grammars a production is in one of the following three forms:

1. $A \to a$
2. $A \to aB$
3. $A \to \varepsilon$

where A and B stand for non-terminals and a for a terminal. The symbol "ε" is used here to indicate "nothing". Some people would write a blank in place of it, but I believe it will be clearer here to put some symbol that stresses the nothingness that is there.

A grammar can be used to generate *sentences* in a *language*. You start with just the initial symbol, and then at each step in a derivation you identify a non-terminal in the current string and use one of the productions to replace it with whatever is to the right of the "$\to$" in a suitable production. The process stops when there are no non-terminals left to replace. Any string of terminals that results is described as a *sentence* generated by the grammar. The set of all possible sentences that a given grammar can generate is known as the *language* that the grammar describes. Some grammars define languages that only have a finite number of sentences in them, others define languages that have an infinite number of sentences[3]. You will find

[2]By saying that it is a *set* I implicitly indicate that any particular production only occurs once in it.

[3]As an exercise you could try to write an ML program that starts with a description of a some grammar and creates a lazy list of all the sentences it can generate. You will not necessarily find this easy, especially if you try to ensure that each possible sentence is included exactly once in the output.

examples of grammars later on in these notes. For now I am just going to report that one of the key results I will prove is that for any regular grammar is is possible to construct a finite automaton that can be used to recognise when sequences of symbols are in the language that the grammar describes. In fact if I look ahead to the Unix Tools course you get towards the end of the year I can imagine that you may hear mentioned particular commonly available unix programs `yacc` and `lex` that are often used when a new programming language must be implemented. The `lex` program represents direct practical use of the ideas I discuss here.

Just as finite automata could be extended to one with unbounded memory, and that led to Part IB material, so phrase structure grammars can exist in more general forms than the ones I limit myself to here. These more general grammars are discussed in the Part 1B course on Compiler Construction.

2.4 Pattern matching

My fourth perspective onto this course is pattern matching, typically in strings of letters. Text editors all provide ways of searching for simple strings in the file being edited. But sometimes it is useful to specify much more elaborate patterns. An idea that has been found to be very convenient to use and quite respectably powerful is to build patterns up in the following way:

1. Any single letter can be used as a pattern. So for instance a or z can be used just on its own to be a pattern that matches just that one literal character in the text being scanned;

2. The pattern λ matches an empty string. Having the ability to cope with this degenerate case is included in the name of completeness and consistency;

3. The pattern $\emptyset$ is a pattern that does not match anything(!), again included for completeness[4];

4. If P_1 and P_2 are two existing patterns then we can write one after the other (P_1P_2) to make a composite pattern that will match anything where the first part of it matches P_1 and the second part matches P_2. The parentheses here are just to stress grouping and are not an essential part of the notation. Note that for any pattern P the patterns $P\lambda$ and λP will be equivalent to just P;

5. If P_1 and P_2 are patterns then $(P_1|P_2)$ is a pattern (the *alternation* of P_1 and

[4]Other presentations of this material might use ε in place of λ or **0** and **1** in place of these two special patterns. I use the symbols I do here so as to be consistent with Sudkamp's book, but you should neither be surprised nor upset if when you find alternative notations in use elsewhere.

P_2) that will match anything that either of its constituents do[5]. For any P we have $P|\emptyset = \emptyset|P = P$;

6. If P is a pattern then P^* is a pattern that matches strings that can be split into zero or more substrings each of which is matched by P. This means[6] that $P^* = \lambda \mid P \mid PP \mid PPP \mid \ldots$. The "*" is known as the Kleene Star, and the expression P^* is the *arbitrary repetition* of P.

Patterns build up in this way are known as *Regular Expressions*, and many text editors and a variety of other Unix (and indeed other operating system) tools use them. You will (of course) come across extended forms of regular expressions which include other operations beyond the ones listed above. I will comment on some such extensions later on. Clearly there are two interesting questions about regular expressions. The first is "What sorts of pattern can they be used to describe?" and the other is "How can we write a program that matches regular expressions against some target text?"

This course answers both of these questions by showing that finite automata, regular grammars and regular expressions are all very closely related: given any one you can derive one of the others and thus describe its behaviour or give a mechanical way of recognising strings that match it. Perhaps the biggest insight that this gives is that a unifying way of looking at things is to view them as *languages*. So we will show that any finite automaton can be completely characterised by the language that it accepts, and that this language can be described by a regular grammar, and that furthermore it is exactly the collection of strings matched by some particular regular expression.

3 The main results that will be proved

In this section I collect a summary of the results that I will prove:

Non-deterministic automata: I will define *non-deterministic finite automata* as an extension of the original sort of finite machine discussed above, and then show that this does not change the range of possible behaviours;

Closure properties and Extended regular expressions: I will look at the intersection and complement of regular languages, and show that they are

[5] Again some people will want to write concatenation as if it had been multiplication, and will use a '$+$' sign where I use '$|$'. Yet others will use '$\cup$' for the "or" operation.

[6] You should be somewhat suspicious any time you see a remark with "…" in it, since there are occasions where infinite formulae have unexpectedly different properties from finite ones. Thus the notation used here is intended as an informal way to help you understand the star operation but care will be needed when we get to formal proofs.

regular. In consequence allowing *and* and *not* operations in the construction of regular expressions gives just a short-hand for expressions that could have been written without the new operators;

Arden's Rule: a result about the language P^*;

For any regular expression there is a regular grammar that defines exactly the same language;

For any regular language there is an automaton that accepts it;

For any automaton there is a regular expression that characterises its behaviour. The fact that automata and regular expressions are so closely linked constitute Kleene's Theorem;

There are non-regular languages, and a result known as the *Pumping Lemma* can often be used to show that a non-regular language is indeed non-regular;

Decision problems: Given two regular expressions or grammars it is possible to decide (systematically and in a finite amount of time) whether they generate the same language. Similarly given two pieces of sequential hardware it is possible (in principle) to decide if their behaviours will be the same;

Some things are hard: Regular languages are one of the nicest, best-behaved sorts of language around. However I will at least be able to describe (if not actually prove) some issues relating to them which show that they are not at all trivial.

Given the equivalence between the languages defined by regular grammars, finite automata and regular expressions it will not end up mattering much which formalism is used as a reference definition of what a regular language is. I will use a definition that a regular language is a language generated by a regular grammar, and prove onwards from there. If somebody else defines a regular language as the language accepted by a finite automaton and then proves that certain grammars define exactly the same classes of language that is OK by me.

4 Regular Grammars

I should start by giving a few examples of regular grammars, together with informal descriptions of the languages that they generate:

1. All possible strings over the alphabet $\{a, b\}$ that are of even length:

$$\begin{array}{rcl} S & \rightarrow & \varepsilon \\ S & \rightarrow & aT \\ S & \rightarrow & bT \\ T & \rightarrow & aS \\ T & \rightarrow & bS \end{array}$$

2. A first attempt at a formal description of what an integer looks like. It insists on a digit followed by an arbitrarily long sequence of additional digits:

$$\begin{array}{rcl} S & \rightarrow & 0D \\ S & \rightarrow & 1D \\ S & \rightarrow & 2D \\ S & \rightarrow & 3D \\ S & \rightarrow & 4D \\ S & \rightarrow & 5D \\ S & \rightarrow & 6D \\ S & \rightarrow & 7D \\ S & \rightarrow & 8D \\ S & \rightarrow & 9D \\ D & \rightarrow & \varepsilon \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 2D \\ D & \rightarrow & 3D \\ D & \rightarrow & 4D \\ D & \rightarrow & 5D \\ D & \rightarrow & 6D \\ D & \rightarrow & 7D \\ D & \rightarrow & 8D \\ D & \rightarrow & 9D \end{array}$$

   You might observe that this way of writing things is somewhat bulky and clumsy. It would be possible to describe a syntax for floating point numbers using this formalism, but perhaps it would not be very convenient.

3. Either the word “sit” or the word “sing”:

$$\begin{array}{rcl} S & \rightarrow & sA \\ S & \rightarrow & sB \\ A & \rightarrow & iC \\ B & \rightarrow & iD \\ C & \rightarrow & t \\ D & \rightarrow & nE \\ E & \rightarrow & g \end{array}$$

   The purpose of this example is to stress that the initial definition of a phrase structure grammar is concerned with *generating* languages and that they are not automatically directly convenient if what you want to do is to *check if a given string is in a language*. Suppose here that your string starts off si then you need some foresight to tell whether the s and the i came by using the productions from S through A or B. In this case you might be able to produce a different grammar that accepts the same language but where you can check input character by character. Is this always the case?

Given a specification of a formal concept (such as that of a “regular grammar”) a natural and important question to ask is how robust the definition is. What happens if the definition is changed a little? Here I will investigate three ways of altering my original definition of a regular grammar and I will show that none of these changes alter the class of languages that can be defined. This will tend to support a claim that this class is a good and natural one to study. Even though the changes I make to the regular grammars leave the class of languages that can be described unaltered that can make a real difference to how easy it can be to construct a grammar that fits a given language. The alterations I will consider are:

- For variant one I disallow productions of the form $A \rightarrow x$, so that only the other two cases are permitted;
- For variant two I add in the possibility of productions such as $A \rightarrow B$ where the right hand side consists of just a single non-terminal. I will refer to such productions as λ-productions;
- The third variation will demand that for any non-terminal A and any terminal x there is exactly one non-terminal B and a production of the form $A \rightarrow x\,B$. In this case there will be a total function (non-terminals$\times$terminals$\rightarrow$non-terminals) that defines which B is associated with any pair (A, x). Productions of the form $A \rightarrow \varepsilon$ are still allowed—the constraint I am applying just

says what happens when a terminal symbol is present on the right hand size of a production. I will describe a grammar that is limited in this way as being *deterministic*, and the total function involved will be called its *transition function*.

In the first of these cases I need to show that any language described by an ordinary regular grammar can be described by a grammar in the restricted form. This is in fact very easy indeed! If the original grammar contained a production $A \rightarrow x$ then a new non-terminal (Q say) can be invented, and the offending rule can be replaced by two new productions:

$$\begin{array}{rcl} A & \rightarrow & x\,Q \\ Q & \rightarrow & \varepsilon \end{array}$$

This simplification of regular grammars may lead to a (very slight) simplification of some proofs that use them.

The next adjustment represent generalisations of my first definition of what a regular grammar is. Allowing λ-productions often makes it easier to construct a grammar that will describe some useful language, and I will illustrate that fact before showing that the extension is not strictly essential.

Consider two regular languages L_1 and L_2 each described by regular grammars (say G_1 and G_2). I will show how to produce a new grammar (using λ-productions) that generates just those strings that could be formed by concatenating a sentence from L_2 onto the end of a sentence from L_1. This construction will then suffice to show that the set of regular languages is closed under the operation of concatenation.

Firstly I note that the names used for non-terminals in a grammar are not very important, so I perform systematic re-naming in my original grammars to ensure that they each end up with different sets of non-terminals. The starting symbol in the first will now be called S_1 and the starting symbol in the second S_2. I also simplify each grammar in accordance with my restriction as described above. Now to join the two grammars together I just find each rule $A \rightarrow \varepsilon$ in G_1 and convert it into a λ-production $A \rightarrow S_2$. I let S_1 be the starting symbol for this adjusted grammar and assert that it can generate exactly all the things that consist of a string from L_1 followed by one from L_2, and that on tracing through a derivation the activation of a production that leads to S_2 shows where (in the string being generated) one part ended and the other starts.

Now to make this useful I need to show that the λ-productions introduced can be removed to leave a grammar in standard form that generates the same language.

I will achieve this by removing λ-productions one at a time. I will start with a grammar G and will search it to find a λ-production $A \rightarrow B$, with A different from B and such that this particular λ-production has not been treated before.

Next I will find all productions with B on their left hand side, which I will write here as $B \rightarrow b_i\ C_i$ (though it should be understood that a case $B \rightarrow \varepsilon$ is also possible. I then extend the original λ-production with a set of new productions of the form $A \rightarrow b_1\ C_i$. It should be clear that any sentence that could have been generated by the original grammar can also be generated by the new one, but without ever activating the λ-production. Note that extending the set of productions will not be allowed to introduce duplicates, and that I am leaving the now-redundant λ-production in there for the while.

Because the set of non-terminals present in a grammar is finite there are only a finite number of possible λ-productions that could possibly be present, so even though the process described above can potentially introduce new ones after a finite amount of work all the ones that there are will have been subject to the above conversion. It is then possible to tidy up the grammar by just removing the λ-productions. Any of the form $A \rightarrow A$ were unimportant anyway (use of them can never help generate a sentence that could not be generated otherwise), while the rest have been expanded out. What will result will be an ordinary regular grammar that generates the same language as the original extended one.

You might like to observe that the above transformation was done in a slightly tricky way. The new productions were added to the grammar but removing the old ones was left until the very end. It was done that way because expanding away one λ-production might generate others, an in particular could re-introduce ones you thought you had got rid of before. Doing things my way avoids such trouble at the cost of keeping a record of which λs have been processed and so that each is only considered once.

My final variation on regular grammars (deterministic ones) is a restriction, so I need to show that any ordinary grammar can be converted into a deterministic one that generates the same language. The motivation for this will become very apparent in the next section, so here I will just describe the recipe to use. This is known as the *power-set construction*. Given a set, its power-set is the set of all subsets. So for example the powerset of $\{a, b, c\}$ is $\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Here I will take the set of non-terminal symbols present in a grammar G and take its power set. I will then show how to construct a new grammar whose non-terminals are the members of this power-set, such that that new grammar is deterministic and it generates the same language as the original language. This is potentially a somewhat painful procedure to apply in practice, since if a set has n elements its power-set will have 2^n. So if the original grammar had say 20 non-terminals the new one will have $2^{20} = 1048576$. The new grammar that I am building will use the same set of terminal symbols as the original. So now I just need to identify a starting symbol and explain what its production rules are.

If S is the starting state of the original grammar I make $\{S\}$ start the new one. Although elsewhere in these notes I will tend to use single letters as names for non-terminals, in the power-set grammar I will write them as sets or sometimes letters in the style A, B standing for such sets. If you like you can re-name them to something neater once the construction has been completed.

Now for the production rules. To make the new grammar deterministic I need to show how how for every possible state and every possible terminal symbol I can give a rule $\mathcal{A} \rightarrow x\ \mathcal{B}_{\mathcal{A},x}$. I will do this by showing what $\mathcal{B}$ to use in each such case. The non-terminals $\mathcal{A}$ and $\mathcal{B}$ will be subsets of the non-terminals in the original grammar. Write this as $\mathcal{A} = \{A_1, A_2, \ldots\}$. Now for each terminal symbol x identify (in the original grammar) each production of the form $A_i \rightarrow x\ B_i$ where A_i is one of the symbols that is a member of $\mathcal{A}$. I then let $\mathcal{B}$ be the set of all the B_i.

Also add a production $\mathcal{A} \rightarrow \varepsilon$ to the new grammar if and only if a production $A_i \rightarrow \varepsilon$ was present in the original (again A_i is a member of $\mathcal{A}$). And that is all there is to it! The construction clearly only involves a finite amount of work[7].

Now I need to show both that the new grammar is deterministic and that it generates exactly the same language as the original one. Well the first of these is easy—its very construction was such that productions that generated new non-terminals did so in a way where the non-terminal on the right hand side was given as a function of the other two symbols involved.

I will be slightly more formal this time about showing that the original and new grammars generate the same language. Firstly I will argue that any sentence that can be generated by the original grammar can also be generated by the new one, and then I will show that the new grammar does not generate any new extraneous sentences.

Suppose we have a sentence generated by the original grammar. Without loss of generality I can suppose that the original grammar does not contain any λ-rules[8]. In such cases any sentence that is n symbols long will be derived by applying just n productions of the form $A \rightarrow x\ B$ plus one of the form $A \rightarrow \varepsilon$. This ability to argue by counting is one of the pay-offs from having shown that λ-productions are not needed. Now I will show that there is a derivation of length n for the same sentence in the new grammar.

Let $\mathcal{Q}_0$ be the state $\{S\}$ that is the starting state of the new grammar. Let x_i be the i-th character of a sentence generated by the original grammar. Observe that in the deterministic grammar we have productions from each state for every possible input symbol, and we can use these to define a unique sequence of states

[7] In quite a few cases a simple-minded use of the above recipe produces a new grammar that has a lot of productions in it that could never possibly by used in any derivation starting from $\{S\}$, and those with tidy minds might like to arrange either not to generate them or to remove them after they have been generated. I will defer issues of optimisation until later on.

[8] If it originally did we could have removed them before starting the real part of the conversion to deterministic form.

$\mathcal{Q}_i$ by demanding that productions $\mathcal{Q}_i \rightarrow x_{i+1}\ \mathcal{Q}_{i+1}$ exists. I then assert that there will also be a production $\mathcal{Q}_n \rightarrow \varepsilon$ present. To see that just check that the definition of the new machine is such that at each stage the state that the original one was in after k productions had been used will be a member of the state $\mathcal{Q}_k$.

Equally if a sentence can be generated by the deterministic grammar there has to be a derivation for it using the original grammar. Here I assert an induction hypothesis. It is that with $\mathcal{Q}_k$ defined as above there are derivations of length k in the original grammar that would lead from S to $x_1\ \ldots\ x_k\ S_k$ for **every** non-terminal (of the original grammar) S_k that is a member of $\mathcal{Q}_k$. The base case $k = 0$ is immediate, and the way that the deterministic grammar was constructed was exactly such as to make the induction step true. Thus the result is true in general. Again adding a consideration of the productions that yield ε finishes the proof.[9]

The main conclusion from all this is that the original definition of regular grammars was probably a good and useful one. The fairly minor adjustments to exactly what sort of productions were permitted left the class of languages that could be described unaltered, as did the more substantial restriction to the deterministic case.

5 Finite Automata

The real reason for introducing deterministic regular grammars is that they are rather obviously equivalent to finite machines, the like of which could be made using flip-flops. Instead of talking about non-terminal symbols I will consider machine states. Instead of terminal symbols I will think of input symbols presented to my machine. And where a regular grammar can have productions from some non-terminals to ε showing that a sentence can stop at some point, I will refer to some of the states of my machine as being *accepting*. Obviously the starting symbol for a grammar corresponds to the state the machine is in when first switched on.

A machine of this sort can be used by feeding it symbols (from its input alphabet, Σ) one at a time. When the machine is in an accepting state this fact is visible[10]. The machine accepts a string of symbols if feeding that string in (starting with the machine in its proper starting state) leads the machine to an accepting state. The language accepted by the machine is just the set of all strings that it would accept.

[9]Observe that I have not written out all the details here—that is left as an exercise for you to do for your supervisor!

[10]I like to think of the machine as having a green light on its top that comes on when the internal configuration is in an accepting state.

Because a machine like this only produces one bit of output, the indication of acceptance, and because it is thus useful to think of its behaviour in terms of the language that it accepts, it is common to refer to the system as a *FDA*. This is an acronym for Finite Deterministic Acceptor.[11]

It should now be clear that given the transition rule for any FDA it would be possible to write out a regular grammar that generated its language. And in view of the result about deterministic regular grammars, it is possible to take **any** regular grammar at all and, after converting it to deterministic form, view it as a description of a piece of hardware. Both of these conversions are ones that you should probably try out on examples, and there will be plenty of exercises to try out in both the text-book and in past exam papers.

The hardware-oriented view of things tends to focus most on recognising strings that form part of a language. You feed the string into a machine and see if it is accepted. The grammar approach is more oriented towards a global description of a language, and provides a way in which you can generate sentences, but in general is less concerned with testing a sentence to see if it is in the some given language. Thus the two views are complementary. The link between them is both a bridge between hardware and software and a piece of practical technology that can be used to help language people test input or hardware people generate all possible behaviours of their systems.

With regular grammars I looked for restrictions and generalisations and found several to study. With regular grammars in mind it proves natural to define an extended sort of finite state machine. This is an *FNA*, or Finite Non-deterministic Acceptor.

Recall that a FDA has an input alphabet (Σ), a set of states (Q), a transition function ($\delta : \Sigma \times Q \to Q$), an initial state and a set of accepting states. An FNA will be very similar but I can generalise in three respects:

The most important one Instead of a transition function I will generalise things to allow an arbitrary relation on $\Sigma \times Q \times Q$ as the specification of what the machine is allowed to do when presented with new input. You might remember from the Discrete Mathematics course that the set of functions $A \to B$ is no more than a subset of the set $(\mathcal{R})(\mathcal{A}, \mathcal{B})$ of all possible relationships on A,B, and so this really is a sensible sort of generalisation to consider. The effect will be that for some symbol-state combinations there may be two or more successor states permitted, and for some there may be none at all;

[11] Again notation in this whole area is a mess. Some people would re-order the words and hence have the acronym as DFA rather than FDA, others would call it a "machine" not just an "acceptor" (DFM), or a "finite state" machine (FSM, DFSM). Generally there is nothing sinister or subtle in all these different notations, there are just lots of different names for the same concept.

optionally I can allow for state-changes in the machine that do not involve processing any input. I can build this into the formalisation by extending my transition relation to be on $(\Sigma \oplus \{\lambda\}) \times Q \times Q$ so that the new pseudo-symbol λ marks a transition that does not involve any real input. This of course corresponds exactly to the use of λ-productions in regular grammars;

less commonly I could specify that instead of having a single definite starting state that the machine had some subset of its states marked as "possible starting states". This gives symmetry between starting and accepting states, and may also be of interest to hardware designers who can not be confident that their system will power up in exactly the state they most wanted.

It is necessary to be a little careful with FNAs, in particular the exact meaning of a string being accepted by an FNA may not at first be obvious. The interpretation that has proved to be useful is that an FNA accepts a string if there is *some* set of transitions of the machine that ends up in an accepting state at the end of the string. Thus the non-determinism or uncertainty does not behave like the uncertainty of most hardware faults where inconsistent behaviour of a machine is unhelpful. On the contrary, it is more useful to think in terms of non-deterministic machines arranging to select from all the transitions available to them just one that will lead to your string being accepted.

A non-deterministic machine may show no possible successor state from some state/symbol combination. This is quite in order and its interpretation is that no string starting that way can ever be accepted.

Delightfully I can just read off results from what I have already proved about regular grammars. Given any language accepted by an FNA (with any or all of the above extensions) it is possible to construct a FDA to accept it. If the FNA had n states then we can certainly construct a FDA with no more than 2^n states. And hardware courses may discuss the issue of state minimisation for such machines and thus lead to more efficient solutions (sometimes).

A point that might be useful to make here is that if a FDA called M has been defined using some particular alphabet Σ then sometime we want a machine that behaves just the same way but which uses some larger alphabet Σ'. If we work informally with automata and just draw pictures to show their configuration and their transitions this does not look as if it is an issue of any substance. However if we check the fine print and now look at the extended machine the transition relationship is not longer a total function, hence we have ended up with an FNA not an FDA.

The main conclusion that emerges is that (at least apart from concerns about efficiency or bulk) it is valid to design a finite machine using the flexibility of non-determinism, because it will always be possible to convert the FNA you design into a FDA later on before you actually try to build it in hardware. In quite a number of cases the design of an FNA will be much easier than going straight to a FDA, so this can be really useful.

The transition function for a FDA will often involve a fairly small alphabet and a fairly small number of states. In such cases it is natural to display it as a table so that the new state of the machine can be looked up in it. This idea leads to a simple and convenient software implementation of FDAs. I will suppose that I represent both symbols and states by integers, and write my code in a mangled and informal version of Modula-3. As an exercise for both this course and the Modula-3 one you might take a specific FDA, code it this way (getting the Modula-3 syntax and other details exactly right) and try it out:

```
integer transition[0..syms,0..states];
bool accepting[0..states];
(* need to initialise the above arrays *)
integer state := 0;
while true do begin
   state := transition[read_symbol(), state];
   if accepting[state] then print "Accept here"
end;
```

Providing the FDA was not so large that the tables needed become ridiculously over-large the above provides a tidy and efficient way of testing if a given string of symbols is accepted by the machine, and thus in the language that it defines.

6 Closure Properties of Regular Languages

Now I have introduced two ways of thinking about regular languages I will prove some more properties that they have. These will show that in a number of obviously sensible operations that one could perform on languages preserve the property of regularity. Sometimes it will be convenient to prove things by reference to a grammar, sometimes by appealing to a FDA, but we have already seen that it is possible to convert between these two forms. In general when I combine two grammars I will want both of them to be defined in terms of the same alphabet. Certainly the constructions mentioned here that use automata will tend not to be properly specified unless this is the case.

6.1 Complementation

If L is a language using the alphabet Σ then I will define $\overline{L}$ as the language consisting of all strings of symbols from Σ that are *not* in L. Note that when you form the complement of a language you must do so relative to an explicit understanding of the alphabet involved. For instance if you just say that L is the language consisting of the single string a then its complement viewing it as a language over the alphabet that consists of just the one symbol a is quite different from the complement over the full English alphabet.

If L is regular then so will $\overline{L}$ be. This would probably not be instantly obvious if you thought in terms of regular grammars. But in terms of FDAs it is easy to show. L is regular and therefore there is a FDA that accepts it. Make a new FDA which is just like this one except that every state that was accepting in the original in non-accepting in the new one, and vice versa. This is still clearly a FDA, and it obviously accepts just those sequences of input symbols that are not in L. Ie it accepts $\overline{L}$. Thus $\overline{L}$ is accepted by some FDA and hence it is regular.

6.2 Union

If L_1 and L_2 are two regular languages, then the union $L_1 \cup L_2$ is also regular. Note that since a language is just a set of strings unions between languages are quite reasonable things to form. I will prove this one two different ways, one using grammars and one using machines, just to show that there are often different ways of demonstrating the same result.

To use grammars, I start by asserting that for L_1 there will be a grammar G_1, and for L_2 I can have G_2, each regular grammars. It is important here that these two grammars have disjoint sets of non-terminals; I will assume this is so here and similarly in other proofs that occur later in these notes. Suppose that the two starting symbols are S_1 and S_2. I form a new grammar whose non-terminals are the disjoint sum of those in the two existing grammars, together with a special new symbol S'. The productions in the new grammar come by just taking all the productions from each of G_1 and G_2, and adding in two additional rules:

$$\begin{array}{rcl} S' & \rightarrow & S_1 \\ S' & \rightarrow & S_2 \end{array}$$

and as you might expect S' becomes the starting symbol for the new grammar. Any string generated by G_1 can be generated using this new grammar by starting with the λ-production $S' \rightarrow S_1$, and equally all strings from L_2 can be produced. And it is a regular grammar (in one of my extended forms). Hence $L_1 \cup L_2$ is generated by a regular grammar and is thus regular.

To use machines I note that there will be FDAs M_1 and M_2 (say) for my two original languages. Form a new machine. Its alphabet will be union of the alphabets used in L_1 and L_2. If M_1 has a set of states Q_1 and similarly for M_2 then the new machines set of states will be $Q_1 \times Q_2$, ie each state will be an ordered pair (q_i, q'_j) with one member of the pair from each machine[12]. Such a state will be accepting if either q_i or q'_j is. The starting state for the new machine will be (q_0, q'_0)

The transition function is then the obvious one got by extending the existing transition functions to the space of ordered pairs—I leave writing down its definition as another exercise. When you have done that it ought to be clear that the new machine enters an accepting state when presented with a string that would have caused either M_1 or M_2 to record acceptance. Again this is sufficient to show that the union language is regular.

6.3 Intersection

Observe that $L \cap M = \overline{\overline{L} \cup \overline{M}}$ and so since we have already proved that complements and unions of regular languages are regular we can deduce that intersections are too.

Alternatively the construction in the previous sub-section that made a machine to accept a union of two languages can easily be modified to accept an intersection instead. All that needs to be changed is the specification of when the product machine accepts a string—it is altered so that a state is only accepting if both q_1 and q_2 are.

6.4 Difference

If the language $L_1 \setminus L_2$ denotes the language of sentences that are in L_1 but not L_2 then it will be regular provided L_1 and L_2 are. This can be seen because $L_1 \setminus L_2 = L_1 \cup \overline{L_2}$, but as for intersection it could be done directly in terms of a product FDA. Note that this is often a nicer operation to work with than raw complementation because the complement of a language introduces an implicit dependency on the alphabet being used.

6.5 Concatenation

Earlier on I showed that if you have two regular languages (at that time defined just by grammars) then the language formed by concatenating them was also regular.

Any finite number of unions, intersections and concatenation operations can be performed on regular languages and the result will remain regular. Infinite numbers of operations in general will not lead to regular results.

[12] I am using q here for states from M_1 and q' for states from M_2.

6.6 Arbitrary repetition

Take a grammar G for a language L, with start-symbol S that generates a language, and add to it the following extra productions: (a) $S \rightarrow \varepsilon$ and (b) whenever $A \rightarrow \varepsilon$ is in the grammar add in (as well) a new production $A \rightarrow S$. The first of these changes ensures that the new grammar can generate zero repetitions of the language L. The second allows the grammar to restart after the end of any sentence, so provides for arbitrary repetition. Because we have still ended up with a regular grammar the resulting language is regular. It is known as L^*.

It might help to explain where this curious notation comes from. In general one writes L^n for n-fold repetition of L. The "$*$" is then used in place of any specific repetition count to indicate arbitrary repetition. It is sometimes convenient to write L^+ for repetition one or more times (where L^* was zero or more repetitions). If L is regular then L^+ is too, as $L^+ = LL^*$.

6.7 Reversal

The language L^R consists of all strings that, when reversed, would be in L. It is what the language L looks like in a mirror. If L is regular then so is L^R. By using my most extended form of FNA I can show this easily. Construct a machine M for L. Then change this by exchanging the sets of starting and accepting states[13], and exchanging the part played by the two states mentioned in the transition relation. The result is (clearly) an FNA that computes backwards relative to the original, and hence accepts L^R. It can then of course be converted to a FDA if you really want.

I will give another (probably neater) proof of this result later on.

7 Regular Expressions

The closure properties of regular languages above lead to another idea for describing them: *regular expressions*. These start from a set of base cases (single symbol languages, the empty language and the language that contains just the empty sentence) and builds up bigger languages using union, concatenation and repetition. Because the base cases are all regular all other languages defined by regular expressions will necessarily be regular too. In a little while I will prove that every regular language can be described this way. The result that regular expressions and regular grammars and FDAs are all equivalent in their expressive power is known as Kleene's Theorem. But first I want to repeat (from section 2.4) the rules

[13] See the section where I introduces FNAa for comments that they can be extended to permit multiple starting states.

for constructing regular expressions, to give a few examples of them and show what a very convenient and compact way of describing languages they provide. If ρ stands for a regular expression, and x for an arbitrary symbol from my alphabet (ie $x \in \Sigma$), then the following rules allow one to build valid regular expressions:

0: $\rho \rightarrow \emptyset$

symbol: $\rho \rightarrow x$

concatenation: $\rho \rightarrow \rho_1\rho_2$

alternation: $\rho \rightarrow \rho_1|\rho_2$

repetition: $\rho \rightarrow \rho_1^*$

Note that $\lambda = \emptyset^*$ so the above rules still allow one to specify a regular expression for the string of length zero.

Here are some examples:

1. Even length strings over $\{a, b\}$ can be described as $((a|b)(a|b))^*$
2. My syntax for integers becomes

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

3. Any string that contains at least three a symbols in a row (over an alphabet a, b, c) is $(a|b|c)^*aaa(a|b|c)^*$.

A formal way of justifying that the collection of strings that match any regular expression forms a regular language will use induction on the size of the regular expression. First observe that a regular expression of size one is just one of the base cases that matches just one string (or in the case of $\emptyset$ does not match any strings at all). It is very easy indeed to exhibit either regular grammars or FDAs to accept these. Now as an induction hypothesis suppose that all regular expressions with size less than k describe regular languages, and consider any expression R of size $k > 1$. By the way regular expressions are built up it will be in one of the three possible forms R_1R_2, $R_1|R_2$ or R_1^* where R_1 and R_2 are smaller regular expressions. But now by the induction hypothesis R_1 and R_2 describe regular languages, and so since R itself is either the concatenation, alternation or arbitrary repetition of these its language is also regular.

The proof of equivalence in the other direction is distinctly harder. I want to show that for any regular grammar or FDA there is a regular expression that describes exactly the same language. The way I will do this involves introducing a further extension of the idea of regular grammars. Recall the the right hand side of a typical production is a single terminal followed by a non-terminal. I will now consider grammars where the right hand size of a production can be a regular expression (in terms of terminals) followed by a single non-terminal. The interpretation of such a grammar is that it will generate all the sentences that could possible arise by both expanding out the production rules in the usual way and by replacing each regular expression by all possible strings that it describes. I will then show how to take an arbitrary (ordinary) regular grammar and convert it into one which consists of a single production $S \rightarrow r$ where r is some (possible rather large) regular expression. I will need to show as I go that the new grammar generates just the same language as the original. In view of the equivalence between regular grammars and automata one could conduct exactly the same proof but describe it in terms of machines, where now the transitions of the machine would be *events* described by regular expressions rather than just occurrences of single input symbols.

Without loss of generality I will demand that my input grammar does not have any λ-production in it.

Firstly I will identify any pairs of productions which share the same two non-terminals, and I will combine them. If I find $A \rightarrow x\ B$ and $A \rightarrow y\ B$ I replace the two productions with one new one $A \rightarrow (x|y)\ B$. I will keep applying this transformation and so can assume henceforth that given any two non-terminals there is at most one production from one to the other.

Now for each non-terminal B (with B different from the start symbol S) I will change the grammar in the following way so as to remove all mention of B.

For the given B find all other non-terminals A and C and regular expressions x, y and z such that $A \neq B$ and $B \neq C$ and the following three productions are present in the grammar:

$$\begin{aligned} A &\rightarrow xB \\ B &\rightarrow yB \\ B &\rightarrow zC \end{aligned}$$

(the case $A = C$ is permitted here) If there are no productions $B \rightarrow y\ B$ then this can be indicated by writing $B \rightarrow \emptyset\ B$ using the regular expression that does not match anything at all! Introduce a new production

$$A \rightarrow xy^*zC$$

Also if $B \rightarrow \varepsilon$ is present introduce

$$A \rightarrow xy^*$$

When this has been done for all possible A and C remove B and all productions involving it. I assert that any sequence of reductions using the original grammar which involve use of the non-terminal B can still be modelled by the new one, and so the language generated has not been altered. But the new grammar has one fewer non-terminal symbol. I will come back to justifying my claim shortly.

By repeating the above transformation any grammar can be reduced to one with only the starting symbol left. The grammar must then be in the form

$$\begin{aligned} S &\rightarrow uS \\ S &\rightarrow v \end{aligned}$$

and now the regular expression u^*v captures it all.

Expressed in a slightly different form the last step is known as Arden's rule. This says that if U and V are languages with U not containing the empty string, and if $L = V \cup UL$ then $L = U^*V$. This rule is what is needed to justify the other steps too. Note that because I did not have any λ-productions in my original grammar all the regular expressions that I produce on the way will only match non-empty strings.

Arden's rule can be proved by induction on the length of strings. Dr Moody's notes for the Diploma Course contain details, but I am not going to work through them here.

Regular expressions are very commonly used to specify patterns to search for in text. Deriving a FDA that accepts the language for such an expression leads to an efficient way of making a computer recognise strings that match the expression, and the matching process does not involve any guesswork or back-tracking. Regular languages are also very widely used in the very early stages of compiling programming languages. They give a convenient way of specifying how the user's program should be split up into tokens. For instance I have already given an example showing the format that can be used to denote an integer. Similarly the rules about what sequences of characters make up valid identifiers, strings or floating point numbers in typical programming languages also tend to be regular. As a (not totally trivial) exercise you could try to construct a good representation for the (regular) language that denotes floating point values. You should expect that such a number either contains a decimal point or an exponent marker (or possibly both), and that the exponent can be signed. But degenerate cases such as ".e-" should not be allowed.

Again as for grammars and machines I will investigate whether there are useful extensions to regular expressions or restrictions on them. Since the regular languages are closed under intersection and complementation I could allow two new ways of building regular expressions, viz $R_1 \;\&\; R_2$ and $\sim R$, to denote the strings that match both R_1 and R_2 and the strings that do not match R. When both these new operations are permitted the resulting class of patters are referred to as *extended regular expressions*. If just the "and" operation is used we have *semi-extended regular expressions*. It may seem that since any language defined by an extended regular expression is still regular that these are frivolous extensions, but they are not. This is because some languages can be described by much more compact extended regular expressions than the shortest possible ordinary regular expression for them. I will discuss this again briefly in a later section on efficiency.

Regular expressions allow one to produce a neat demonstration that the reversal of a regular language is regular. Exhibit a regular expression that matches the language. Then if this is a base case leave it alone, if it is $A_1|A_2$ replace it with $A_1^R|A_2^R$, if it is A_1A_2 replace it with $A_2^RA_1^R$ and if it is A^* replace it with $(A^R)^*$. The result will (clearly) be a regular expression that matches its reversal, and hence that language is regular.

In the other direction there has been a study of what happens if you restrict the rules that can be used to build regular expressions. If you permit intersection and negation but do not allow use of the repetition operator "$*$" the collection of strings the can be generated are known as the *star-free sets*. There are regular languages that are not star-free, but even that result is not as instantly easy to prove as you might have hoped, and I will not justify it here. Leading on from that is the concept of the *star depth* of a regular language. Any regular language can be described by an extended regular expression. Its star depth will be the depth of nesting of stars in the least-nested expression for it. So the star-free sets are just those languages with star depth zero. The question of whether there are any regular languages whose star depth is greater than 1 appears to lead one into amazingly deep and murky waters!

8 Non-Regular languages and the Pumping Lemma

Thus far I have concentrated on nice positive results that show that certain languages **are** regular, and that certain ways of combining existing languages lead to regular results. Now I want to give some concrete examples of languages that are **not** regular, and explain one of the most commonly-used ways of taking a specific language and proving that it can not possibly be regular. This is known as the *Pumping Lemma*.

I will illustrate the method behind the Pumping Lemma by considering the language $\{a^nb^n\}$, ie all those strings that consist on a bunch of a symbols followed by an equal number of b symbols. I assert that this is not regular.

If it were, then there would exist some FDA to accept it. Suppose this machine has N states. Now consider the behaviour of the FDA when it is presented with the string a^Nb^N, and in particular the sequence of states that it passes through as it sees the first N input symbols. Because the machine only has N states and (if you include the starting state that it is in before it has seen anything) there are $N + 1$ states it must pass through in this calculation, some state must me a repeat of a previous one. This deep (!) result is generally known as the pigeon-hole principle. If you try to post $N + 1$ items into N pigeon-holes at least one hole must end up with two or more items in it. Now the repeated state means that the machine performed some operations and got back to a place it had been before. That little loop could be repeated any number of times (including zero times) and the machine would be equally happy. So now we imagine carrying on the calculation until the machine accepts out sample input, and then we tinker with the number of times it goes round the loop. The result will be that we exhibit a bunch of extra strings that it will accept. But we have chosen our set-up so that all these strings have the same number of b symbols but different numbers of a symbols, and hence most of them are not in the language that we were supposed to be accepting. The only way out of this difficulty is to conclude that it was not possible to have a finite machine that accepted our language, and hence the language can not be regular after all.

The Pumping Lemma states: For every regular language L, there is some number $k \geq 1$ such that all strings $w \in L$ with length(w) $\geq k$ there is some way of expressing w as a concatenation $w = u_1 v u_2$ that has the properties

1. length(v) ≥ 1;
2. length($u_1 v$) $\leq k$;
3. for all $n \geq 0$, $u_1 v^n u_2 \in L$.

Two things deserve stress here. The Lemma tells us that the number k exists, but does not tell us what it is. It also says that the decomposition of w exists as described—it certainly does not even pretend to say that any decomposition of w into three parts $w = u_1 v u_2$ will satisfy the three properties. Just that there is at least one such decomposition.

The proof is much as that for the particular case I worked through. Given any regular L there must be a FDA that will accept it, and I then suppose that there is such a machine with k states. Now properties (1) and (2) just say that while processing the first k symbols of an input string w this machine must repeat a state. u_1 is the string of symbols read in before the repeated state is reached for the first time, and v is the non-empty set of symbols that cause the machine to traverse an internal loop. Property (2) says that the loop must be completed by the time k symbols have been seen. Property (3) is just the assertion that the machine will then accept strings obtained by traversing the loop any number (n) times.

Here is a demonstration of the use of the Pumping Lemma. I assert that the language $\{a^p\}$ with p a prime number is not a regular language. In other words no finite machine could cause a light to flash each time the number of input symbols it had been given was prime. First observe that there are an infinite number of primes, so for any k associated with my language I could select a prime p with $p > 2k$. Now the pumping lemma explains that the string $w = a^{2p}$ can be split up into three parts u_1, v and u_2 with some length constraints. length($u_1 v$) $\leq k$, and I will call that length r, and length(v) ≥ 1, and I will call that s. Thus

$$\begin{aligned} u_1 &= a^{r-s} \\ v &= a^s \\ u_2 &= a^{p-r} \end{aligned}$$

Now I will consider the string $u_1 v^{p-s} u_2$ which the pumping lemma assures me will be accepted by the automaton. Its length is $(r-s) + s(p-s) + (p-r) = (s+1)(p-s)$. Verifying that is just simple algebra! But now because s is at least 1, $s+1$ is at least 2, and furthermore $p-s$ must also be at least 2 (because $s \leq k$, $k \geq 1$ and $p > 2k$). Hence I have a string whose length is certainly not prime. The only conclusion is that the language concerned could not have been regular after all.

An analogous proof show that the language of palindromes is not regular (a palindrome is something that is equal to its own reversal. Consider sentences like "Able was I ere I saw Elba" and "Madam, I'm Adam", but ignore some whitespace and punctuation for these English language examples). Similarly for all strings where brackets nest properly, and hence for pretty well all programming languages.

It is perhaps useful to observe that the Pumping Lemma could have been reasoned about in terms of regular grammars, and the magic number k would then have been the number of non-terminal symbols used in the grammar. At later stages in the CST when you come across more general sorts of phrase structure grammars you should be aware that they will come with their own more general Pumping Lemmas which again seek to capture the insight that with only a finite number of non-terminal symbols there must eventually be some sort of potential for repetition present in a language specification.

9 Decision Problems for regular languages

Given a regular language and a particular string we can decide if the string is in the language. For instance a FDA for the language can be constructed and the string can be fed to it—if at the end of the string the FDA is in an accepting state the string is in the given language. The test can be performed in a systematic way that will always terminate with a clear-cut answer to our question. There are a number of other questions about regular languages that can be answered using systematic procedures, although I will explain later on that sometimes the amount of work involved gets out of hand.

1. Is a regular language empty? For instance you might describe a regular language by some elaborate extended regular expression and then wonder if there are any strings in it at all. To decide the problem construct a FDA for the language. If this has k states then if the machine accepts anything at all it will accept a string of length less than k. To prove this, suppose to the contrary that the shortest string accepted was of length $\geq k$, then the pumping lemma applies and shows that there will be a shorter accepted string. If the alphabet Σ provides just m distinct symbols there are m^k strings of length k, and we can (at least in principle) check each of these to see if on of them or one of the initial substrings of one of them is accepted. If none are then the language is empty.

2. Are two regular languages identical? This can be resolved by observing that two languages are identical if their symmetric difference $L_1 \setminus L_2 \cup L_2 \setminus L_2$ is empty. Set differences and unions of regular languages are regular, so this case reduces to the previous one.

Apart from the fact that the procedures described are horribly costly the results might mean that detecting if there will be any input sequence that gets a piece of sequential hardware into a given state can be tested for, and verifying that two different hardware designs will operate identically can also be verified. Realistic techniques for hardware verification will have to wait until a Part II course!

10 Efficiency

Here I will show that it is possible to have regular languages with small FDAs such that their concatenation needs a large FDA. The languages I will use to illustrate this are $L_1 = (a|b)^*$ and $L_2 = a(a|b)^n$ for various values of n. First observe that L_1 is easily implemented using a machine with just one state, while L_2 is equally easily recognised using around n states. Now the language $L = L_1 L_2$ is one where the n-th symbol from the end of any string must be an a. I assert that any FDA that accepts this language must have at least 2^n states.

To prove this I will consider the 2^n strings that start with a and then have all the possible length-n sequences of a and b symbols. Presenting each of these to the FDA must leave it in an accepting state. Now I will argue that these must be 2^n *different* accepting states. Suppose that two of them were the same, then there would be two different sequences leading to that state. These two sequences would differ in some particular symbol—one would have an a where the other had a b. Suppose that place of difference was k from the start, then feed the FDA k more symbols (which might as well all be a). Since it is deterministic the state that it arrives in is definitely well-defined. Considering whether this state is accepting or not leads to a proof of my assertion.

You might suspect from this that the exponential blow-up seen here was just because I started with and insisted on ending with deterministic automata. Maybe working with and FNA (or correspondingly a regular grammar) would mean that all the useful operations on languages could be done without causing severe increase in the size of a description. One counter-example here will be to look at the intersection operation. Let p be a prime number. Each of the languages $L_2 = (a^2)^*$, $L_3 = (a^3)^*$, $\ldots L_p = (a^p)^*$ can be characterised by an automaton with at worst p states. Determinism or non-determinism plain does not make a big difference here since DFAs do the job as neatly as one could hope. Now let N be the product of the primes 2, 3, $\ldots p$: the intersection of all these languages is clearly the language $(a^N)^*$ Thus the shortest non-empty string it can accept is of length N. Hence any automaton that accepts it (deterministic or not) must have at least N states.[14] N grows very rapidly as a function of the aggregate size of the input automata.

A final comment on performance and practicality relates to extended regular expressions. Suppose that R is an extended regular expression that can be written in n symbols, we know that there will be some ordinary (non-extended) regular expression R' that describes the same language. Perhaps R' will need to be rather larger than R and it would be nice to have a bound on the expansion possible.

Define

$$\begin{aligned} H_0(n) &= b \\ H_i(n) &= 2^{H_{i-1}(n)} \end{aligned}$$

so that $H_k(n)$ involves a tower k high of exponentiation. Then a result that is too hard to prove at this stage shows that the size of R' expressed as a function of R can grow faster than H_k for any fixed k you care to choose. This is pretty desperate! Since I can not complete the proof here I should leave you with an exercise that relates to one early part of it:

10.1 Exercise:

Define the n-th ruler sequence as one that starts off $0102010401020108 01\ldots$. Every other character will be a 0. Of the remaining characters every other will be a 1. After filtering out the 0s and 1s every other character is a 2. And so on, until

[14]Exercise: write out a formal justification of this.

depth n where characters n just repeat. Such a sequence naturally repeats after just 2^n characters. Show how to make a semi-extended regular expression (ie one that allows "and" operations) that is tolerably short but which describes one of these ruler sequences.

11 Conclusions

The main lessons that can be drawn from this course are as follows:

1. Regular Expressions, Regular Grammars, FDAs and FNAs can all be interpreted as just different ways of thinking about a single set of languages, known as the *regular languages*. If you start with any one form of description there will be a systematic way of converting it into any of the others;

2. Regular languages have good closure properties. But beware that all the closure results I have quoted are for *finite* combinations of languages. Infinite intersections or unions of regular languages may lead to languages that are not regular;

3. A pigeon-hole principle and the Pumping Lemma allow us to show that some languages are not regular. If you have a regular language then many of the questions you might want to ask have answers that at least in theory could be found my applying systematic procedures. But sometimes the costs would in reality be utterly outside all practicable bounds.

I hope you have also been convinced that the results explained here were not all intuitively obvious in advance and that many of them are entertaining and decorative.

References

[1] Thomas A Sudkamp. *Languages and Machines*. Addison Wesley, 1988.

Thanks are due to Alan Mycroft for helpful comments on an earlier draft of these notes.