N

Lecture Notes on

Regular Languages and Finite Automata

for Part IA of the Computer Science Tripos

Prof. Andrew M. Pitts
Cambridge University Computer Laboratory

First Edition 1998.
Revised 1999, 2000, 2001, 2002, 2003.

Contents

Learning Guide

The notes are designed to accompany six lectures on regular languages and finite automata for Part IA of the Cambridge University Computer Science Tripos. The aim of this short course will be to introduce the mathematical formalisms of finite state machines, regular expressions and grammars, and to explain their applications to computer languages. As such, it covers some basic theoretical material which Every Computer Scientist Should Know. Direct applications of the course material occur in the various CST courses on compilers. Further and related developments will be found in the CST Part IB courses **Computation Theory** and **Semantics of Programming Languages** and the CST Part II course **Topics in Concurrency**.

This course contains the kind of material that is best learned through practice. The books mentioned below contain a large number of problems of varying degrees of difficulty, and some contain solutions to selected problems. A few exercises are given at the end of each section of these notes and relevant past Tripos questions are indicated there. At the end of the course students should be able to explain how to convert between the three ways of representing regular sets of strings introduced in the course; and be able to carry out such conversions by hand for simple cases. They should also be able to prove whether or not a given set of strings is regular.

Recommended books Textbooks which cover the material in this course also tend to cover the material you will meet in the CST Part IB courses on **Computation Theory** and **Complexity Theory**, and the theory underlying parsing in various courses on compilers. There is a large number of such books. Three recommended ones are listed below.

- D. C. Kozen, *Automata and Computability* (Springer-Verlag, New York, 1997).
- T. A. Sudkamp, *Languages and Machines* (Addison-Wesley Publishing Company, Inc., 1988).
- J. E. Hopcroft, R. Motwani and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation, Second Edition* (Addison-Wesley, 2001).

Note The material in these notes has been drawn from several different sources, including the books mentioned above and previous versions of this course by the author and by others. Any errors are of course all the author's own work. A list of corrections will be available from the course web page (follow links from www.cl.cam.ac.uk/Teaching/). A lecture(r) appraisal form is included at the end of the notes. Please take time to fill it in and return it. Alternatively, fill out an electronic version of the form via the URL www.cl.cam.ac.uk/cgi-bin/lr/login.

Andrew Pitts
Andrew.Pitts@cl.cam.ac.uk

1 Regular Expressions

Doubtless you have used pattern matching in the command-line shells of various operating systems (Slide 1) and in the search facilities of text editors. Another important example of the same kind is the 'lexical analysis' phase in a compiler during which the text of a program is divided up into the allowed tokens of the programming language. The algorithms which implement such pattern-matching operations make use of the notion of a *finite automaton* (which is Greeklish for *finite state machine*). This course reveals (some of!) the beautiful theory of finite automata (yes, that is the plural of 'automaton') and their use for recognising when a particular string matches a particular pattern.

Pattern matching

What happens if, at a Unix/Linux shell prompt, you type

`ls *`

and press return?

Suppose the current directory contains files called `regfla.tex`, `regfla.aux`, `regfla.log`, `regfla.dvi`, and (strangely) `.aux`. What happens if you type

`ls *.aux`

and press return?

Slide 1

1.1 Alphabets, strings, and languages

The purpose of Section 1 is to introduce a particular language for patterns, called *regular expressions*, and to formulate some important problems to do with pattern-matching which will be solved in the subsequent sections. But first, here is some notation and terminology to do with character strings that we will be using throughout the course.

Alphabets

An ***alphabet*** is specified by giving a finite set, Σ, whose elements are called ***symbols***. For us, any set qualifies as a possible alphabet, so long as it is finite.

Examples:
$\Sigma_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ — 10-element set of decimal digits.
$\Sigma_2 = \{a, b, c, \ldots, x, y, z\}$ — 26-element set of lower-case characters of the English language.
$\Sigma_3 = \{S \mid S \subseteq \Sigma_1\}$ — 2^{10}-element set of all subsets of the alphabet of decimal digits.

Non-example:
$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ — set of all non-negative whole numbers is not an alphabet, because it is infinite.

Slide 2

Strings over an alphabet

A ***string of length*** n (≥ 0) over an alphabet Σ is just an ordered n-tuple of elements of Σ, written without punctuation.

Example: if $\Sigma = \{a, b, c\}$, then a, ab, aac, and $bbac$ are strings over Σ of lengths one, two, three and four respectively.

$\boxed{\Sigma^* \stackrel{\text{def}}{=} \text{set of all strings over } \Sigma \text{ of any finite length.}}$

N.B. there is a unique string of length zero over Σ, called the ***null string*** (or ***empty string***) and denoted $\boxed{\varepsilon}$ (no matter which Σ we are talking about).

Slide 3

Concatenation of strings

The ***concatenation*** of two strings $u, v \in \Sigma^*$ is the string uv obtained by joining the strings end-to-end.

Examples: If $u = ab$, $v = ra$ and $w = cad$, then $vu = raab$, $uu = abab$ and $wv = cadra$.

This generalises to the concatenation of three or more strings. E.g. $uvwuv = abracadabra$.

Slide 4

Slides 2 and 3 define the notions of an ***alphabet*** Σ, and the set Σ^* of finite ***strings*** over an alphabet. The length of a string u will be denoted by $length(u)$. Slide 4 defines the operation of ***concatenation*** of strings. We make no notational distinction between a symbol $a \in \Sigma$ and the corresponding string of length one over Σ: so Σ can be regarded as a subset of Σ^*. Note that Σ^* is never empty—it always contains the ***null string***, ε, the unique string of length zero. Note also that for any $u, v, w \in \Sigma^*$

$$u\varepsilon = u = \varepsilon u \quad \text{and} \quad (uv)w = uvw = u(vw)$$

and $length(uv) = length(u) + length(v)$.

Example 1.1.1. Examples of Σ^* for different Σ:

(i) If $\Sigma = \{a\}$, then Σ^* contains

$$\varepsilon, a, aa, aaa, aaaa, \dots$$

(ii) If $\Sigma = \{a, b\}$, then Σ^* contains

$$\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots$$

(iii) If $\Sigma = \emptyset$ (the ***empty set*** — the unique set with no elements), then $\Sigma^* = \{\varepsilon\}$, the set just containing the null string.

1.2 Pattern matching

Slide 5 defines the patterns, or ***regular expressions***, over an alphabet Σ that we will use. Each such regular expression, r, represents a whole set (possibly an infinite set) of strings in Σ^* that ***match*** r. The precise definition of this matching relation is given on Slide 6. It might seem odd to include a regular expression $\emptyset$ that is matched by no strings at all—but it is technically convenient to do so. Note that the regular expression ε is in fact equivalent to $\emptyset^*$, in the sense that a string u matches $\emptyset^*$ iff it matches ε (iff $u = \varepsilon$).

Regular expressions over an alphabet Σ

- each symbol $a \in \Sigma$ is a regular expression
- ε is a regular expression
- $\emptyset$ is a regular expression
- if r and s are regular expressions, then so is $(r|s)$
- if r and s are regular expressions, then so is rs
- if r is a regular expression, then so is $(r)^*$

Every regular expression is built up inductively, by *finitely many* applications of the above rules.

(N.B. we assume ε, $\emptyset$, $($, $)$, $|$, and * are *not* symbols in Σ.)

Slide 5

Remark 1.2.1 (Binding precedence in regular expressions). In the definition on Slide 5 we assume implicitly that the alphabet Σ does not contain the six symbols

$$\varepsilon \quad \emptyset \quad (\quad) \quad | \quad ^*$$

Then, concretely speaking, the regular expressions over Σ form a certain set of strings over the alphabet obtained by adding these six symbols to Σ. However it makes things more readable if we adopt a slightly more abstract syntax, dropping as many brackets as possible and using the convention that

$-^*$ binds more tightly than $- -$, binds more tightly than $-|-$.

So, for example, $r|st^*$ means $(r|s(t)^*)$, not $(r|s)(t)^*$, or $((r|st))^*$, etc.

Matching strings to regular expressions

- u matches $a \in \Sigma$ iff $u = a$
- u matches ε iff $u = \varepsilon$
- no string matches $\emptyset$
- u matches $r|s$ iff u matches either r or s
- u matches rs iff it can be expressed as the concatenation of two strings, $u = vw$, with v matching r and w matching s
- u matches r^* iff either $u = \varepsilon$, or u matches r, or u can be expressed as the concatenation of two or more strings, each of which matches r

Slide 6

The definition of 'u matches r^*' on Slide 6 is equivalent to saying

> for some $n \geq 0$, u can be expressed as a concatenation of n strings, $u = u_1u_2 \ldots u_n$, where each u_i matches r.

The case $n = 0$ just means that $u = \varepsilon$ (so ε always matches r^*); and the case $n = 1$ just means that u matches r (so any string matching r also matches r^*). For example, if $\Sigma = \{a, b, c\}$ and $r = ab$, then the strings matching r^* are

$$\varepsilon, ab, abab, ababab, \text{etc.}$$

Note that we didn't include a regular expression for the '$*$' occurring in the UNIX examples on Slide 1. However, *once we know which alphabet we are referring to*, $\Sigma = \{a_1, a_2, \ldots, a_n\}$ say, we can get the effect of $*$ using the regular expression

$$(a_1|a_2|\ldots|a_n)^*$$

which is indeed matched by any string in Σ^* (because $a_1|a_2|\ldots|a_n$ is matched by any symbol in Σ).

Examples of matching, with $\Sigma = \{0, 1\}$

- $0|1$ is matched by each symbol in Σ
- $1(0|1)^*$ is matched by any string in Σ^* that starts with a '1'
- $((0|1)(0|1))^*$ is matched by any string of even length in Σ^*
- $(0|1)^*(0|1)^*$ is matched by any string in Σ^*
- $(\varepsilon|0)(\varepsilon|1)|11$ is matched by just the strings ε, 0, 1, 01, and 11
- $\emptyset 1|0$ is just matched by 0

Slide 7

Notation 1.2.2. The notation $r + s$ is quite often used for what we write as $r|s$.

The notation r^n, for $n \geq 0$, is an abbreviation for the regular expression obtained by concatenating n copies of r. Thus:

$$\begin{cases} r^0 & \stackrel{\text{def}}{=} \varepsilon \\ r^{n+1} & \stackrel{\text{def}}{=} r(r^n). \end{cases}$$

Thus u matches r^* iff u matches r^n for some $n \geq 0$.

We use r^+ as an abbreviation for rr^*. Thus u matches r^+ iff it can be expressed as the concatenation of *one or more* strings, each one matching r.

1.3 Some questions about languages

Slide 8 defines the notion of a ***formal language*** over an alphabet. We take a very extensional view of language: a formal language is completely determined by the 'words in the dictionary', rather than by any grammatical rules. Slide 9 gives some important questions about languages, regular expressions, and the matching relation between strings and regular expressions.

Languages

A (formal) ***language*** L over an alphabet Σ is just a set of strings in Σ^*.

Thus any subset $L \subseteq \Sigma^*$ determines a language over Σ.

The ***language determined by a regular expression*** r over Σ is

$$L(r) \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid u \text{ matches } r\}.$$

Two regular expressions r and s (over the same alphabet) are ***equivalent*** iff $L(r)$ and $L(s)$ are equal sets (i.e. have exactly the same members).

Slide 8

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r (over the same alphabet), computes whether or not u matches r?

(b) In formulating the definition of regular expressions, have we missed out some practically useful notions of pattern?

(c) Is there an algorithm which, given two regular expressions r and s (over the same alphabet), computes whether or not they are equivalent? (Cf. Slide 8.)

(d) Is every language of the form $L(r)$?

Slide 9

The answer to question (a) on Slide 9 is 'yes'. Algorithms for deciding such pattern-matching questions make use of finite automata. We will see this during the next few sections.

If you have used the UNIX utility `grep`, or a text editor with good facilities for regular expression based search, like `emacs`, you will know that the answer to question (b) on Slide 9 is also 'yes'—the regular expressions defined on Slide 5 leave out some forms of pattern that one sees in such applications. However, the answer to the question is also 'no', in the sense that (for a fixed alphabet) these extra forms of regular expression are definable, up to equivalence, from the basic forms given on Slide 5. For example, if the symbols of the alphabet are ordered in some standard way, it is common to provide a form of pattern for naming ranges of symbols—for example $[\mathtt{a} - \mathtt{z}]$ might denote a pattern matching any lower-case letter. It is not hard to see how to define a regular expression (albeit a rather long one) which achieves the same effect. However, some other commonly occurring kinds of pattern are much harder to describe using the rather minimalist syntax of Slide 5. The principle example is ***complementation***, $\sim(r)$:

$$u \text{ matches } \sim(r) \quad \text{iff} \quad u \textit{ does not } \text{match } r.$$

It will be a corollary of the work we do on finite automata (and a good measure of its power) that every pattern making use of the complementation operation $\sim(-)$ can be replaced by an equivalent regular expression just making use of the operations on Slide 5. But why do we stick to the minimalist syntax of regular expressions on that slide? The answer is that it reduces the amount of work we will have to do to show that, in principle, matching strings against patterns can be decided via the use of finite automata.

The answer to question (c) on Slide 9 is 'yes' and once again this will be a corollary of the work we do on finite automata. (See Section 5.3.)

Finally, the answer to question (d) is easily seen to be 'no', provided the alphabet Σ contains at least one symbol. For in that case Σ^* is countably infinite; and hence the number of languages over Σ, i.e. the number of subsets of Σ^* is uncountable. (Recall Cantor's diagonal argument.) But since Σ is a finite set, there are only countably many regular expressions over Σ. (Why?) So the answer to (d) is 'no' for cardinality reasons. However, even amongst the countably many languages that are 'finitely describable' (an intuitive notion that we will not formulate precisely) many are not of the form $L(r)$ for any regular expression r. For example, in Section 5.2 we will use the 'Pumping Lemma' to see that

$$\{a^n b^n \mid n \geq 0\}$$

is not of this form.

1.4 Exercises

Exercise 1.4.1. Write down an ML data type declaration for a type constructor `'a regExp` whose values correspond to the regular expressions over an alphabet `'a`.

Exercise 1.4.2. Find regular expressions over $\{0, 1\}$ that determine the following languages:

(a) $\{u \mid u \text{ contains an even number of 1's}\}$

(b) $\{u \mid u \text{ contains an odd number of 0's}\}$

Exercise 1.4.3. For which alphabets Σ is the set Σ^* of all finite strings over Σ itself an alphabet?

Exercise 1.4.4. If an alphabet Σ contains only one symbol, what does Σ^* look like? (Answer: see Example 1.1.1(i).) So if $\Sigma = \{\varepsilon\}$ is the set consisting of just the null string, what is Σ^*? (The answer is *not* $\{\varepsilon\}$.) Moral: the punctuation-free notation we use for the concatenation of two strings can sometimes be confused with the punctuation-free notation we use to denote strings of individual symbols.

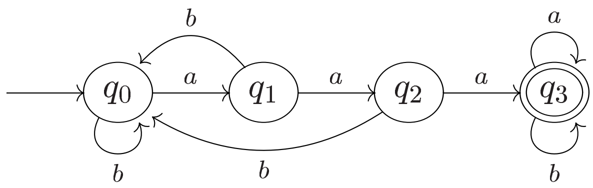
Tripos questions 1999.2.1(s) 1997.2.1(q) 1996.2.1(i) 1993.5.12

2 Finite State Machines

We will be making use of mathematical models of physical systems called *finite automata*, or *finite state machines* to recognise whether or not a string is in a particular language. This section introduces this idea and gives the precise definition of what constitutes a finite automaton. We look at several variations on the definition (to do with the concept of determinism) and see that they are equivalent for the purpose of recognising whether or not a string is in a given language.

2.1 Finite automata

Example of a finite automaton



States: q_0, q_1, q_2, q_3.

Input symbols: a, b.

Transitions: as indicated above.

Start state: q_0.

Accepting state(s): q_3.

Slide 10

The key features of this abstract notion of 'machine' are listed below and are illustrated by the example on Slide 10.

- There are only finitely many different ***states*** that a finite automaton can be in. In the example there are four states, labelled q_0, q_1, q_2, and q_3.

- We do not care at all about the internal structure of machine states. All we care about is which ***transitions*** the machine can make between the states. A symbol from some fixed alphabet Σ is associated with each transition: we think of the elements of Σ as ***input symbols***. Thus all the possible transitions of the finite automaton can be specified by giving a finite graph whose vertices are the states and whose edges have both a direction and a label (drawn from Σ). In the example $\Sigma = \{a, b\}$ and the only possible transitions from state q_1 are

$$q_1 \xrightarrow{b} q_0 \quad \text{and} \quad q_1 \xrightarrow{a} q_2.$$

In other words, in state q_1 the machine can either input the symbol b and enter state q_0, or it can input the symbol a and enter state q_2. (Note that transitions from a state back to the same state are allowed: e.g. $q_3 \xrightarrow{a} q_3$ in the example.)

- There is a distinguished ***start state***.[1] In the example it is q_0. In the graphical representation of a finite automaton, the start state is usually indicated by means of a unlabelled arrow.

- The states are partitioned into two kinds: ***accepting states***[2] and non-accepting states. In the graphical representation of a finite automaton, the accepting states are indicated by double circles round the name of each such state, and the non-accepting states are indicated using single circles. In the example there is only one accepting state, q_3; the other three states are non-accepting. (The two extreme possibilities that *all* states are accepting, or that *no* states are accepting, are allowed; it is also allowed for the start state to be accepting.)

The reason for the partitioning of the states of a finite automaton into 'accepting' and 'non-accepting' has to do with the use to which one puts finite automata—namely to recognise whether or not a string $u \in \Sigma^*$ is in a particular language ($=$ subset of Σ^*). Given u we begin in the start state of the automaton and traverse its graph of transitions, using up the symbols in u in the correct order reading the string from left to right. If we can use up all the symbols in u in this way and reach an accepting state, then u is in the language 'accepted' (or 'recognised') by this particular automaton; otherwise u is not in that language. This is summed up on Slide 11.

[1] The term ***initial state*** is a common synonym for 'start state'.
[2] The term ***final state*** is a common synonym for 'accepting state'.

$L(M)$, ***language accepted*** **by a finite automaton** M

consists of all strings u over its alphabet of input symbols satisfying $q_0 \xrightarrow{u}{}^{*} q$ with q_0 the start state and q some accepting state. Here

$$q_0 \xrightarrow{u}{}^{*} q$$

means, if $u = a_1 a_2 \dots a_n$ say, that for some states $q_1, q_2, \dots, q_n = q$ (not necessarily all distinct) there are transitions of the form

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n = q.$$

N.B.
case $n = 0$: $q \xrightarrow{\varepsilon}{}^{*} q'$ iff $q = q'$
case $n = 1$: $q \xrightarrow{a}{}^{*} q'$ iff $q \xrightarrow{a} q'$.

Slide 11

Example 2.1.1. Let M be the finite automaton pictured on Slide 10. Using the notation introduced on Slide 11 we have:

$$q_0 \xrightarrow{aaab}{}^{*} q_3 \qquad \text{(so } aaab \in L(M))$$
$$q_0 \xrightarrow{abaa}{}^{*} q \quad \text{iff} \quad q = q_2 \qquad \text{(so } abaa \notin L(M))$$
$$q_2 \xrightarrow{baaa}{}^{*} q \quad \text{iff} \quad q = q_3 \qquad \text{(no conclusion about } L(M)).$$

In fact in this case

$$L(M) = \{u \mid u \text{ contains three consecutive } a\text{'s}\}.$$

(For q_i $(i = 0, 1, 2)$ corresponds to the state in the process of reading a string in which the last i symbols read were all a's.) So $L(M)$ coincides with the language $L(r)$ determined by the regular expression

$$r = (a|b)^* aaa(a|b)^*$$

(cf. Slide 8).

A ***non-deterministic finite automaton*** (NFA), M,
is specified by

- a finite set $States_M$ (of ***states***)
- a finite set Σ_M (the alphabet of ***input symbols***)
- for each $q \in States_M$ and each $a \in \Sigma_M$, a subset $\Delta_M(q, a) \subseteq States_M$ (the set of states that can be reached from q with a single ***transition*** labelled a)
- an element $s_M \in States_M$ (the ***start state***)
- a subset $Accept_M \subseteq States_M$ (of ***accepting states***)

Slide 12

2.2 Determinism, non-determinism, and ε-transitions

Slide 12 gives the formal definition of the notion of finite automaton. Note that the function Δ_M gives a precise way of specifying the allowed transitions of M, via: $q \xrightarrow{a} q' \quad \text{iff} \quad q' \in \Delta_M(q, a)$.

The reason for the qualification 'non-deterministic' on Slide 12 is because in general, for each state $q \in States_M$ and each input symbol $a \in \Sigma_M$, we allow the possibilities that there are no, one, or many states that can be reached in a single transition labelled a from q, corresponding to the cases that $\Delta_M(q, a)$ has no, one, or many elements. For example, if M is the NFA pictured on Slide 13, then

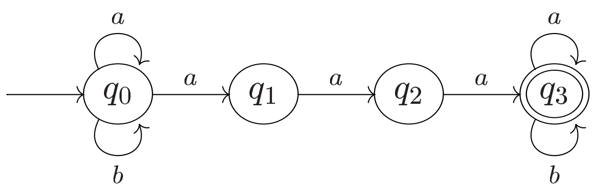
$\Delta_M(q_1, b) = \emptyset$ i.e. in M, no state can be reached from q_1 with a transition labelled b;

$\Delta_M(q_1, a) = \{q_2\}$ i.e. in M, precisely one state can be reached from q_1 with a transition labelled a;

$\Delta_M(q_0, a) = \{q_0, q_1\}$ i.e. in M, precisely two states can be reached from q_0 with a transition labelled a.

Example of a non-deterministic finite automaton

Input alphabet: $\{a, b\}$.
States, transitions, start state, and accepting states as shown:



The language accepted by this automaton is the same as for the automaton on Slide 10, namely

$$\{u \in \{a, b\}^* \mid u \text{ contains three consecutive } a\text{'s}\}.$$

Slide 13

When each subset $\Delta_M(q, a)$ has exactly one element we say that M is ***deterministic***. This is a particularly important case and is singled out for definition on Slide 14.

The finite automaton pictured on Slide 10 is deterministic. But note that if we took the same graph of transitions but insisted that the alphabet of input symbols was $\{a, b, c\}$ say, then we have specified an NFA not a DFA, since for example $\Delta_M(q_0, c) = \emptyset$. The moral of this is: *when specifying an NFA, as well as giving the graph of state transitions, it is important to say what is the alphabet of input symbols* (because some input symbols may not appear in the graph at all).

When constructing machines for matching strings with regular expressions (as we will do in Section 3) it is useful to consider finite state machines exhibiting an 'internal' form of non-determinism in which the machine is allowed to change state without consuming any input symbol. One calls such transitions ***ε-transitions*** and writes them as

$$q \xrightarrow{\varepsilon} q'.$$

This leads to the definition on Slide 15. Note that in an NFA$^\varepsilon$, M, *we always assume that ε is not an element of the alphabet Σ_M of input symbols.*

A ***deterministic finite automaton*** (DFA)
is an NFA M with the property that for each $q \in States_M$ and $a \in \Sigma_M$, the finite set $\Delta_M(q, a)$ contains exactly one element—call it $\delta_M(q, a)$.

Thus in this case transitions in M are essentially specified by a ***next-state function***, δ_M, mapping each (state, input symbol)-pair (q, a) to the unique state $\delta_M(q, a)$ which can be reached from q by a transition labelled a:

$$q \xrightarrow{a} q' \quad \text{iff} \quad q' = \delta_M(q, a)$$

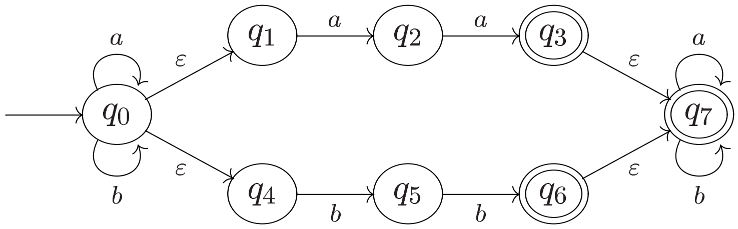
Slide 14

An ***NFA with ε-transitions*** (NFA$^\varepsilon$)
is specified by an NFA M together with a binary relation, called the ***ε-transition relation***, on the set $States_M$. We write

$$q \xrightarrow{\varepsilon} q'$$

to indicate that the pair of states (q, q') is in this relation.

Example (with input alphabet = $\{a, b\}$):



Slide 15

$L(M)$, *language accepted* by an NFA$^\varepsilon$ M

consists of all strings u over the alphabet Σ_M of input symbols satisfying $q_0 \stackrel{u}{\Rightarrow} q$ with q_0 the initial state and q some accepting state. Here $\cdot \stackrel{-}{\Rightarrow} \cdot$ is defined by:

$q \stackrel{\varepsilon}{\Rightarrow} q'$ iff $q = q'$ or there is a sequence $q \stackrel{\varepsilon}{\rightarrow} \cdots q'$ of one or more ε-transitions in M from q to q'

$q \stackrel{a}{\Rightarrow} q'$ (for $a \in \Sigma_M$) iff $q \stackrel{\varepsilon}{\Rightarrow} \cdot \stackrel{a}{\rightarrow} \cdot \stackrel{\varepsilon}{\Rightarrow} q'$

$q \stackrel{ab}{\Rightarrow} q'$ (for $a, b \in \Sigma_M$) iff $q \stackrel{\varepsilon}{\Rightarrow} \cdot \stackrel{a}{\rightarrow} \cdot \stackrel{\varepsilon}{\Rightarrow} \cdot \stackrel{b}{\rightarrow} \cdot \stackrel{\varepsilon}{\Rightarrow} q'$

and similarly for longer strings

Slide 16

When using an NFA$^\varepsilon$ M to accept a string $u \in \Sigma^*$ of input symbols, we are interested in sequences of transitions in which the symbols in u occur in the correct order, but with zero or more ε-transitions before or after each one. We write

$$q \stackrel{u}{\Rightarrow} q'$$

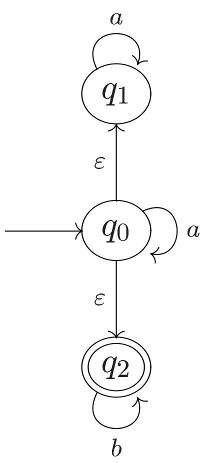
to indicate that such a sequence exists from state q to state q' in the NFA$^\varepsilon$. Then, by definition u ***is accepted by the NFA$^\varepsilon$*** M iff $q_0 \stackrel{u}{\Rightarrow} q$ holds for q_0 the start state and q some accepting state: see Slide 16. For example, for the NFA$^\varepsilon$ on Slide 15, it is not too hard to see that the language accepted consists of all strings which either contain two consecutive a's or contain two consecutive b's, i.e. the language determined by the regular expression $(a|b)^*(aa|bb)(a|b)^*$.

2.3 A subset construction

Note that every DFA is an NFA (whose transition relation is deterministic) and that every NFA is an NFA$^\varepsilon$ (whose ε-transition relation is empty). It might seem that non-determinism and ε-transitions allow a greater range of languages to be characterised as recognisable by a finite automaton, but this is not so. We can use a construction, called the ***subset construction***, to convert an NFA$^\varepsilon$ M into a DFA PM accepting the same language (at the expense of increasing the number of states, possibly exponentially). Slide 17 gives an example of this construction. The name 'subset construction' refers to the fact that there is one state of PM for each subset of the set $States_M$ of states of M. Given two subsets $S, S' \subseteq States_M$, there is a transition $S \stackrel{a}{\rightarrow} S'$ in PM just in case S' consists of all the M-states q' reachable from states q in S via the $\cdot \overset{a}{\Rightarrow} \cdot$ relation defined on Slide 16, i.e. such that we can get from q to q' in M via finitely many ε-transitions followed by an a-transition followed by finitely many ε-transitions.

Example of the subset construction

M :



δ_{PM} :	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_1\}$	$\{q_1\}$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_1, q_2\}$	$\{q_1\}$	$\{q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$

Slide 17

By definition, the start state of PM is the subset of $States_M$ whose elements are the states reachable by ε-transitions from the start state of M; and a subset $S \subseteq States_M$ is an accepting state of PM iff some accepting state of M is an element of S. Thus in the example on Slide 17 the start state is $\{q_0, q_1, q_2\}$ and

$$
\begin{array}{lll}
ab \in L(M) & \text{because in } M\text{:} & q_0 \xrightarrow{a} q_0 \xrightarrow{\varepsilon} q_2 \xrightarrow{b} q_2 \\
ab \in L(PM) & \text{because in } PM\text{:} & \{q_0, q_1, q_2\} \xrightarrow{a} \{q_0, q_1, q_2\} \xrightarrow{b} \{q_2\}.
\end{array}
$$

Indeed, in this case $L(M) = L(a^*b^*) = L(PM)$. The fact that M and PM accept the same language in this case is no accident, as the Theorem on Slide 18 shows. That slide also gives the definition of the subset construction in general.

Theorem. For each NFA$^\varepsilon$ M there is a DFA PM with the same alphabet of input symbols and accepting exactly the same strings as M, i.e. with $L(PM) = L(M)$

Definition of PM (refer to Slide 12):

- $States_{PM} \stackrel{\text{def}}{=} \{S \mid S \subseteq States_M\}$
- $\Sigma_{PM} \stackrel{\text{def}}{=} \Sigma_M$
- $S \xrightarrow{a} S'$ in PM iff $S' = \delta_{PM}(S, a)$, where $\delta_{PM}(S, a) \stackrel{\text{def}}{=} \{q' \mid \exists q \in S\,(q \stackrel{a}{\Rightarrow} q' \text{ in } M)\}$
- $s_{PM} \stackrel{\text{def}}{=} \{q \mid s_M \stackrel{\varepsilon}{\Rightarrow} q\}$
- $Accept_{PM} \stackrel{\text{def}}{=}$ $\{S \in States_{PM} \mid \exists q \in S\,(q \in Accept_M)\}$

Slide 18

To prove the theorem on Slide 18, given any NFA$^\varepsilon$ M we have to show that $L(M) = L(PM)$. We split the proof into two halves.

Proof that $L(M) \subseteq L(PM)$. Consider the case of ε first: if $\varepsilon \in L(M)$, then $s_M \stackrel{\varepsilon}{\Rightarrow} q$ for some $q \in Accept_M$, hence $s_{PM} \in Accept_{PM}$ and thus $\varepsilon \in L(PM)$. Now given any non-null string $u = a_1 a_2 \ldots a_n$, if u is accepted by M then there is a sequence of transitions in M of the form

(1) $$s_M \stackrel{a_1}{\Rightarrow} q_1 \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} q_n \in Accept_M.$$

Since it is deterministic, feeding $a_1 a_2 \ldots a_n$ to PM results in the sequence of transitions

(2) $$s_{PM} \xrightarrow{a_1} S_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} S_n$$

where $S_1 = \delta_{PM}(s_{PM}, a_1)$, $S_2 = \delta_{PM}(S_1, a_2)$, etc. By definition of δ_{PM} (Slide 18), from (1) we deduce

$$\begin{aligned} q_1 &\in \delta_{PM}(s_{PM}, a_1) = S_1 \\ \text{so } q_2 &\in \delta_{PM}(S_1, a_2) = S_2 \\ &\ldots \\ \text{so } q_n &\in \delta_{PM}(S_{n-1}, a_n) = S_n \end{aligned}$$

and hence $S_n \in Accept_{PM}$ (because $q_n \in Accept_M$). Therefore (2) shows that u is accepted by PM. □

Proof that $L(PM) \subseteq L(M)$. Consider the case of ε first: if $\varepsilon \in L(PM)$, then $s_{PM} \in Accept_{PM}$ and so there is some $q \in s_{PM}$ with $q \in Accept_M$, i.e. $s_M \overset{\varepsilon}{\Rightarrow} q \in Accept_M$ and thus $\varepsilon \in L(M)$. Now given any non-null string $u = a_1 a_2 \ldots a_n$, if u is accepted by PM then there is a sequence of transitions in PM of the form (2) with $S_n \in Accept_{PM}$, i.e. with S_n containing some $q_n \in Accept_M$. Now since $q_n \in S_n = \delta_{PM}(S_{n-1}, a_n)$, by definition of δ_{PM} there is some $q_{n-1} \in S_{n-1}$ with $q_{n-1} \overset{a_n}{\Rightarrow} q_n$ in M. Then since $q_{n-1} \in S_{n-1} = \delta_{PM}(S_{n-2}, a_{n-1})$, there is some $q_{n-2} \in S_{n-2}$ with $q_{n-2} \overset{a_{n-1}}{\Rightarrow} q_{n-1}$. Working backwards in this way we can build up a sequence of transitions like (1) until, at the last step, from the fact that $q_1 \in S_1 = \delta_{PM}(s_{PM}, a_1)$ we deduce that $s_M \overset{a_1}{\Rightarrow} q_1$. So we get a sequence of transitions (1) with $q_n \in Accept_M$, and hence u is accepted by M. □

2.4 Summary

The important concepts in Section 2 are those of a *deterministic finite automaton* (DFA) and the language of strings that it accepts. Note that if we know that a language L is of the form $L = L(M)$ for some DFA M, then we have a method for deciding whether or not any given string u (over the alphabet of L) is in L or not: *begin in the start state of M and carry out the sequence of transitions given by reading u from left to right* (at each step the next state is uniquely determined because M is deterministic); *if the final state reached is accepting, then u is in L, otherwise it is not.*

We also introduced other kinds of finite automata (with non-determinism and ε-transitions) and proved that they determine exactly the same class of languages as DFAs.

2.5 Exercises

Exercise 2.5.1. For each of the two languages mentioned in Exercise 1.4.2 find a DFA that accepts exactly that set of strings.

Exercise 2.5.2. The example of the subset construction given on Slide 17 constructs a DFA with eight states whose language of accepted strings happens to be $L(a^*b^*)$. Give a DFA with the same language of accepted strings, but fewer states. Give an NFA with even fewer states that does the same job.

Exercise 2.5.3. Given a DFA M, construct a new DFA M' with the same alphabet of input symbol Σ_M and with the property that for all $u \in \Sigma_M^*$, u is accepted by M' iff u is not accepted by M.

Exercise 2.5.4. Given two DFAs M_1, M_2 with the same alphabet Σ of input symbols, construct a third such DFA M with the property that $u \in \Sigma^*$ is accepted by M iff it is accepted by both M_1 and M_2. [Hint: take the states of M to be ordered pairs (q_1, q_2) of states with $q_1 \in States_{M_1}$ and $q_2 \in States_{M_2}$.]

Tripos questions 2001.2.1(d) 2000.2.1(b) 1998.2.1(s) 1995.2.19 1992.4.9(a)

3 Regular Languages, I

Slide 19 defines the notion of a ***regular language***, which is a set of strings of the form $L(M)$ for some DFA M (cf. Slides 11 and 14). The slide also gives the statement of Kleene's Theorem, which connects regular languages with the notion of matching strings to regular expressions introduced in Section 1: the collection of regular languages coincides with the collection of languages determined by matching strings with regular expressions. The aim of this section is to prove part (a) of Kleene's Theorem. We will tackle part (b) in Section 4.

Definition

A language is ***regular*** iff it is the set of strings accepted by some deterministic finite automaton.

Kleene's Theorem

(a) *For any regular expression* r, $L(r)$ *is a regular language* (cf. Slide 8).

(b) *Conversely, every regular language is the form* $L(r)$ *for some regular expression* r.

Slide 19

3.1 Finite automata from regular expressions

Given a regular expression r, over an alphabet Σ say, we wish to construct a DFA M with alphabet of input symbols Σ and with the property that for each $u \in \Sigma^*$, u matches r iff u is accepted by M—so that $L(r) = L(M)$.

Note that by the Theorem on Slide 18 it is enough to construct an NFA$^\varepsilon$ N with the property $L(N) = L(r)$. For then we can apply the subset construction to N to obtain a DFA $M = PN$ with $L(M) = L(PN) = L(N) = L(r)$. Working with finite automata that are non-deterministic and have ε-transitions simplifies the construction of a suitable finite automaton from r.

Let us fix on a particular alphabet Σ and from now on only consider finite automata whose set of input symbols is Σ. The construction of an NFA$^\varepsilon$ for each regular expression r over Σ proceeds by recursion on the syntactic structure of the regular expression, as follows.

(i) For each atomic form of regular expression, a ($a \in \Sigma$), ε, and $\emptyset$, we give an NFA$^\varepsilon$ accepting just the strings matching that regular expression.

(ii) Given any NFA$^\varepsilon$s M_1 and M_2, we construct a new NFA$^\varepsilon$, $\mathit{Union}(M_1, M_2)$ with the property

$$L(\mathit{Union}(M_1, M_2)) = \{u \mid u \in L(M_1) \text{ or } u \in L(M_2)\}.$$

Thus $L(r_1|r_2) = L(\mathit{Union}(M_1, M_2))$ when $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$.

(iii) Given any NFA$^\varepsilon$s M_1 and M_2, we construct a new NFA$^\varepsilon$, $\mathit{Concat}(M_1, M_2)$ with the property

$$L(\mathit{Concat}(M_1, M_2)) = \{u_1 u_2 \mid u_1 \in L(M_1) \text{ and } u_2 \in L(M_2)\}.$$

Thus $L(r_1 r_2) = L(\mathit{Concat}(M_1, M_2))$ when $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$.

(iv) Given any NFA$^\varepsilon$ M, we construct a new NFA$^\varepsilon$, $\mathit{Star}(M)$ with the property

$$L(\mathit{Star}(M)) = \{u_1 u_2 \ldots u_n \mid n \geq 0 \text{ and each } u_i \in L(M)\}.$$

Thus $L(r^*) = L(\mathit{Star}(M))$ when $L(r) = L(M)$.

Thus starting with step (i) and applying the constructions in steps (ii)–(iv) over and over again, we eventually build NFA$^\varepsilon$s with the required property for every regular expression r.

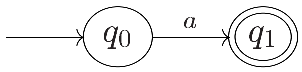
Put more formally, one can prove the statement

for all $n \geq 0$, and for all regular expressions of size $\leq n$, there exists an NFA$^\varepsilon$ M such that $L(r) = L(M)$

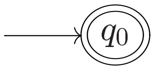
by mathematical induction on n, using step (i) for the base case and steps (ii)–(iv) for the induction steps. Here we can take the *size* of a regular expression to be the number of occurrences of union ($-|-$), concatenation ($-\,-$), or star ($-^*$) in it.

Step (i) Slide 20 gives NFAs whose languages of accepted strings are respectively $L(a) = \{a\}$ (any $a \in \Sigma$), $L(\varepsilon) = \{\varepsilon\}$, and $L(\emptyset) = \emptyset$.

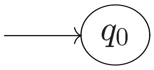
NFAs for atomic regular expressions



just accepts the one-symbol string a



just accepts the null string, ε



accepts no strings

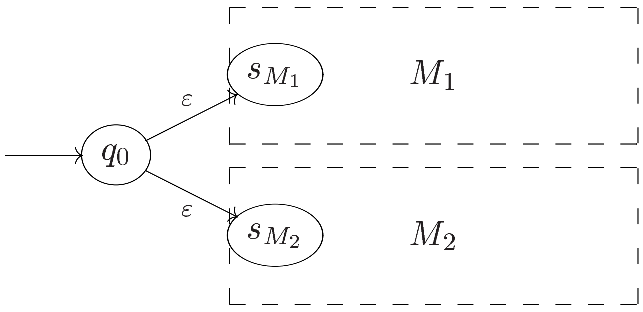
Slide 20

Step (ii) Given NFA$^\varepsilon$s M_1 and M_2, the construction of $Union(M_1, M_2)$ is pictured on Slide 21. First, renaming states if necessary, we assume that $States_{M_1}$ and $States_{M_2}$ are disjoint. Then the states of $Union(M_1, M_2)$ are all the states in either M_1 or M_2, together with a new state, called q_0 say. The start state of $Union(M_1, M_2)$ is this q_0 and its accepting states are all the states that are accepting in either M_1 or M_2. Finally, the transitions of $Union(M_1, M_2)$ are given by all those in either M_1 or M_2, together with two new ε-transitions out of q_0 to the start states of M_1 and M_2.

Thus if $u \in L(M_1)$, i.e. if we have $s_{M_1} \stackrel{u}{\Rightarrow} q_1$ for some $q_1 \in Accept_{M_1}$, then we get $q_0 \stackrel{\varepsilon}{\to} s_{M_1} \stackrel{u}{\Rightarrow} q_1$ showing that $u \in L(Union(M_1, M_2))$. Similarly for M_2. So $L(Union(M_1, M_2))$ contains the union of $L(M_1)$ and $L(M_2)$. Conversely if u is accepted by $Union(M_1, M_2)$, there is a transition sequence $q_0 \stackrel{u}{\Rightarrow} q$ with $q \in Accept_{M_1}$ or $q \in Accept_{M_2}$. Clearly, in either case this transition sequence has to begin with one or other of the ε-transitions from q_0, and thereafter we get a transition sequence entirely in one or other of M_1 or M_2 finishing in an acceptable state for that one. So if $u \in L(Union(M_1, M_2))$, then either $u \in L(M_1)$ or $u \in L(M_2)$. So we do indeed have

$$L(Union(M_1, M_2)) = \{u \mid u \in L(M_1) \text{ or } u \in L(M_2)\}.$$

$Union(M_1, M_2)$



Set of accepting states is union of $Accept_{M_1}$ and $Accept_{M_2}$.

Slide 21

Step (iii) Given NFA$^\varepsilon$s M_1 and M_2, the construction of $Concat(M_1, M_2)$ is pictured on Slide 22. First, renaming states if necessary, we assume that $States_{M_1}$ and $States_{M_2}$ are disjoint. Then the states of $Concat(M_1, M_2)$ are all the states in either M_1 or M_2. The start state of $Concat(M_1, M_2)$ is the start state of M_1. The accepting states of $Concat(M_1, M_2)$ are the accepting states of M_2. Finally, the transitions of $Concat(M_1, M_2)$ are given by all those in either M_1 or M_2, together with new ε-transitions from each accepting state of M_1 to the start state of M_2 (only one such new transition is shown in the picture).

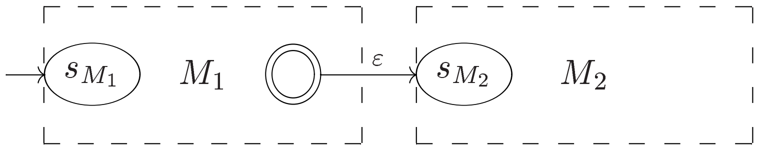
Thus if $u_1 \in L(M_1)$ and $u_2 \in L(M_2)$, there are transition sequences $s_{M_1} \stackrel{u_1}{\Rightarrow} q_1$ in M_1 with $q_1 \in Accept_{M_1}$, and $s_{M_2} \stackrel{u_2}{\Rightarrow} q_2$ in M_2 with $q_2 \in Accept_{M_2}$. These combine to yield

$$s_{M_1} \stackrel{u_1}{\Rightarrow} q_1 \stackrel{\varepsilon}{\rightarrow} s_{M_2} \stackrel{u_2}{\Rightarrow} q_2$$

in $Concat(M_1, M_2)$ witnessing the fact that $u_1 u_2$ is accepted by $Concat(M_1, M_2)$. Conversely, it is not hard to see that every $v \in L(Concat(M_1, M_2))$ is of this form. For any transition sequence witnessing the fact that v is accepted starts out in the states of M_1 but finishes in the disjoint set of states of M_2. At some point in the sequence one of the new ε-transitions occurs to get from M_1 to M_2 and thus we can split v as $v = u_1 u_2$ with u_1 accepted by M_1 and u_2 accepted by M_2. So we do indeed have

$$L(Concat(M_1, M_2)) = \{u_1 u_2 \mid u_1 \in L(M_1) \text{ and } u_2 \in L(M_2)\}.$$

$Concat(M_1, M_2)$



Set of accepting states is $Accept_{M_2}$.

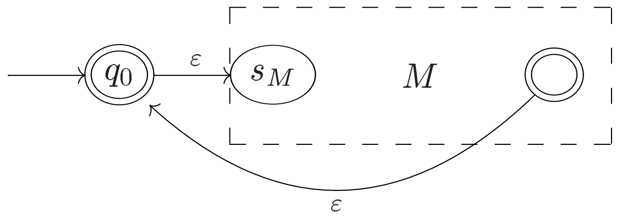
Slide 22

Step (iv) Given an NFA$^\varepsilon$ M, the construction of $Star(M)$ is pictured on Slide 23. The states of $Star(M)$ are all those of M together with a new state, called q_0 say. The start state of $Star(M)$ is q_0 and this is also the only accepting state of $Star(M)$. Finally, the transitions of $Star(M)$ are all those of M together with new ε-transitions from q_0 to the start state of M and from each accepting state of M to q_0 (only one of this latter kind of transition is shown in the picture).

Clearly, $Star(M)$ accepts ε (since its start state is accepting) and any concatenation of one or more strings accepted by M. Conversely, if v is accepted by $Star(M)$, the occurrences of q_0 in a transition sequence witnessing this fact allow us to split v into the concatenation of zero or more strings, each of which is accepted by M. So we do indeed have

$$L(Star(M)) = \{u_1 u_2 \ldots u_n \mid n \geq 0 \text{ and each } u_i \in L(M)\}.$$

$Star(M)$



The only accepting state of $Star(M)$ is q_0.

Slide 23

This completes the proof of part (a) of Kleene's Theorem (Slide 19). Figure 1 shows how the step-by-step construction applies in the case of the regular expression $(a|b)^*a$ to produce an NFA$^\varepsilon$ M satisfying $L(M) = L((a|b)^*a)$. Of course an automaton with fewer states and ε-transitions doing the same job can be crafted by hand. The point of the construction is that it provides an automatic way of producing automata for any given regular expression.

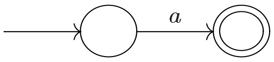
3.2 Decidability of matching

The proof of part (a) of Kleene's Theorem provides us with a positive answer to question (a) on Slide 9. In other words, it provides a method that, given any string u and regular expression r, decides whether or not u matches r. The method is:

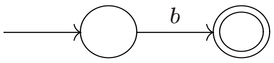
- construct a DFA M satisfying $L(M) = L(r)$;
- beginning in M's start state, carry out the sequence of transitions in M corresponding to the string u, reaching some state q of M (because M is deterministic, there is a unique such transition sequence);
- check whether q is accepting or not: if it is, then $u \in L(M) = L(r)$, so u matches r; otherwise $u \notin L(M) = L(r)$, so u does not match r .

Note. The subset construction used to convert the NFA$^\varepsilon$ resulting from steps (i)–(iv) of Section 3.1 to a DFA produces an exponential blow-up of the number of states. (PM has

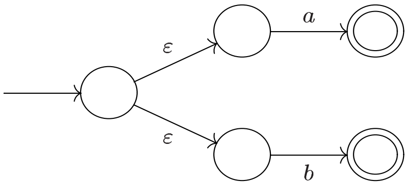
Step of type (i): a



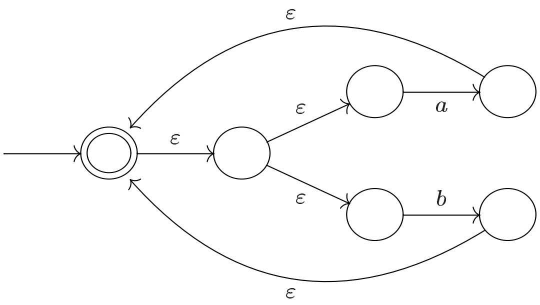
Step of type (i): b



Step of type (ii): $a|b$



Step of type (iv): $(a|b)^*$



Step of type (iii): $(a|b)^*a$

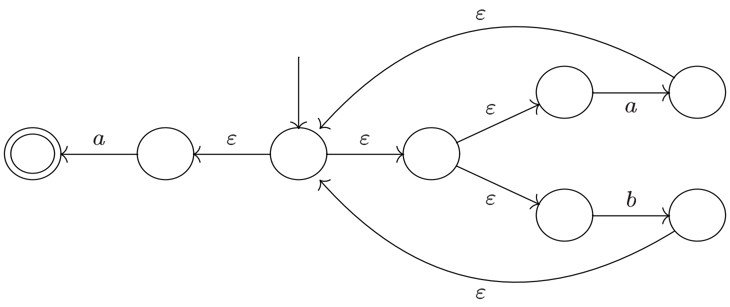


Figure 1: Steps in constructing an NFA$^\varepsilon$ for $(a|b)^*a$

2^n states if M has n.) This makes the method described above very inefficient. (Much more efficient algorithms exist.)

3.3 Exercises

Exercise 3.3.1. Why can't the automaton $Star(M)$ required in step (iv) of Section 3.1 be constructed simply by taking M and adding new ε-transitions back from each accepting state to its start state?

Exercise 3.3.2. Work through the steps in Section 3.1 to construct an NFA$^\varepsilon$ M satisfying $L(M) = L((\varepsilon|b)^*aab^*)$. Do the same for some other regular expressions.

Exercise 3.3.3. Show that any finite set of strings is a regular language.

Tripos question 1992.4.9(b) [needs Slide 26 from Section 4]

4 Regular Languages, II

The aim of this section is to prove part (b) of Kleene's Theorem (Slide 19).

4.1 Regular expressions from finite automata

Given any DFA M, we have to find a regular expression r (over the alphabet of input symbols of M) satisfying $L(r) = L(M)$. In fact we do something more general than this, as described in the Lemma on Slide 24.[1] Note that if we can find such regular expressions $r^Q_{q,q'}$ for any choice of Q, q, and q', then the problem is solved. For taking Q to be the whole of $States_M$ and q to be the start state, s say, then by definition of $r^Q_{s,q'}$, a string u matches this regular expression iff there is a transition sequence $s \xrightarrow{u}{}^* q'$ in M. As q' ranges over the finitely many accepting states, $q_1, \ldots, q_k$ say, then we match exactly all the strings accepted by M. In other words the regular expression $r^Q_{s,q_1}|\cdots|r^Q_{s,q_k}$ has the property we want for part (b) of Kleene's Theorem. (In case $k = 0$, i.e. there are *no* accepting states in M, then $L(M)$ is empty and so we can use the regular expression $\emptyset$.)

Lemma *Given an NFA M, for each subset $Q \subseteq States_M$ and each pair of states $q, q' \in States_M$, there is a regular expression $r^Q_{q,q'}$ satisfying*

$$L(r^Q_{q,q'}) = \{u \in (\Sigma_M)^* \mid q \xrightarrow{u}{}^* q' \text{ in } M \text{ with all intermediate states of the sequence in } Q\}.$$

Hence $L(M) = L(r)$, where $r = r_1|\cdots|r_k$ and
$k =$ number of accepting states,
$r_i = r^Q_{s,q_i}$ with $Q = States_M$,
$s =$ start state,
$q_i = i$th accepting state.

(In case $k = 0$, take r to be the regular expression $\emptyset$.)

Slide 24

Proof of the Lemma on Slide 24. The regular expression $r^Q_{q,q'}$ can be constructed by induction on the number of elements in the subset Q.

[1]The lemma works just as well whether M is deterministic or non-deterministic; it also works for NFA$^\varepsilon$s, provided we replace $\xrightarrow{u}{}^*$ by $\overset{u}{\Rightarrow}$ (cf. Slide 16).

Base case, Q is empty. In this case, for each pair of states q, q', we are looking for a regular expression to describe the set of strings

$$\{u \mid q \xrightarrow{u}{}^{*} q' \text{ with } \textit{no} \text{ intermediate states}\}.$$

So each element of this set is either a single input symbol a (if $q \xrightarrow{a} q'$ holds in M) or possibly ε, in case $q = q'$. If there are no input symbols that take us from q to q' in M, we can simply take

$$r^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } q \neq q' \\ \varepsilon & \text{if } q = q'. \end{cases}$$

On the other hand, if there are some such input symbols, $a_1, \ldots, a_k$ say, we can take

$$r^{\emptyset}_{q,q'} \stackrel{\text{def}}{=} \begin{cases} a_1|\cdots|a_k & \text{if } q \neq q' \\ a_1|\cdots|a_k|\varepsilon & \text{if } q = q'. \end{cases}$$

Induction step. Suppose we have defined the required regular expressions for all subsets of states with n elements. If Q is a subset with $n+1$ elements, choose some element $q_0 \in Q$ and consider the n-element set $Q \setminus \{q_0\} = \{q \in Q \mid q \neq q_0\}$. Then for any pair of states $q, q' \in \mathit{States}_M$, by inductive hypothesis we have already constructed the regular expressions

$$r_1 \stackrel{\text{def}}{=} r^{Q\setminus\{q_0\}}_{q,q'}, \quad r_2 \stackrel{\text{def}}{=} r^{Q\setminus\{q_0\}}_{q,q_0}, \quad r_3 \stackrel{\text{def}}{=} r^{Q\setminus\{q_0\}}_{q_0,q_0}, \quad \text{and} \quad r_4 \stackrel{\text{def}}{=} r^{Q\setminus\{q_0\}}_{q_0,q'}.$$

Consider the regular expression

$$r \stackrel{\text{def}}{=} r_1|r_2(r_3)^*r_4.$$

Clearly every string matching r is in the set

$$\{u \mid q \xrightarrow{u}{}^{*} q' \text{ with all intermediate states in this sequence in } Q\}.$$

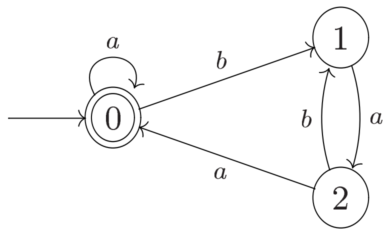
Conversely, if u is in this set, consider the number of times the sequence of transitions $q \xrightarrow{u}{}^{*} q'$ passes through state q_0. If this number is zero then $u \in L(r_1)$ (by definition of r_1). Otherwise this number is $k \geq 1$ and the sequence splits into $k+1$ pieces: the first piece is in $L(r_2)$ (as the sequence goes from q to the first occurrence of q_0), the next $k-1$ pieces are in $L(r_3)$ (as the sequence goes from one occurrence of q_0 to the next), and the last piece is in $L(r_4)$ (as the sequence goes from the last occurrence of q_0 to q'). So in this case u is in $L(r_2(r_3)^*r_4)$. So in either case u is in $L(r)$. So to complete the induction step we can define $r^{Q}_{q,q'}$ to be this regular expression $r = r_1|r_2(r_3)^*r_4$. $\square$

4.2 An example

Perhaps an example will help to understand the rather clever argument in Section 4.1. The example will also demonstrate that we do not have to pursue the inductive construction of the regular expression to the bitter end (the base case $Q = \emptyset$): often it is possible to find some of the regular expressions $r^{Q}_{q,q'}$ one needs by *ad hoc* arguments.

Note also that at the inductive steps in the construction of a regular expression for M we are free to choose which state q_0 to remove from the current state set Q. A good rule of thumb is: *choose a state that disconnects the automaton as much as possible.*

Example



Direct inspection yields:

$r^{\{0\}}_{i,j}$	0	1	2
0			
1	$\emptyset$	ε	a
2	aa^*	$b\mid aa^*b$	ε

$r^{\{0,2\}}_{i,j}$	0	1	2
0	a^*	a^*b	
1			
2			

Slide 25

As an example, consider the NFA shown on Slide 25. Since the start state is 0 and this is also the only accepting state, the language of accepted strings is that determined by the regular expression $r^{\{0,1,2\}}_{0,0}$. Choosing to remove state 1 from the state set, we have

(3) $$L(r^{\{0,1,2\}}_{0,0}) = L(r^{\{0,2\}}_{0,0} | r^{\{0,2\}}_{0,1} (r^{\{0,2\}}_{1,1})^* r^{\{0,2\}}_{1,0}).$$

Direct inspection shows that $L(r^{\{0,2\}}_{0,0}) = L(a^*)$ and $L(r^{\{0,2\}}_{0,1}) = L(a^*b)$. To calculate $L(r^{\{0,2\}}_{1,1})$, and $L(r^{\{0,2\}}_{1,0})$, we choose to remove state 2:

$$L(r^{\{0,2\}}_{1,1}) = L(r^{\{0\}}_{1,1} | r^{\{0\}}_{1,2} (r^{\{0\}}_{2,2})^* r^{\{0\}}_{2,1})$$
$$L(r^{\{0,2\}}_{1,0}) = L(r^{\{0\}}_{1,0} | r^{\{0\}}_{1,2} (r^{\{0\}}_{2,2})^* r^{\{0\}}_{2,0}).$$

These regular expressions can all be determined by inspection, as shown on Slide 25. Thus

$$L(r^{\{0,2\}}_{1,1}) = L(\varepsilon | a(\varepsilon)^*(b|aa^*b))$$

and it's not hard to see that this is equal to $L(\varepsilon | aa^*b)$; and

$$L(r^{\{0,2\}}_{1,0}) = L(\emptyset | a(\varepsilon)^*(aa^*))$$

which is equal to $L(aaa^*)$. Substituting all these values into (3), we get

$$L(r_{0,0}^{\{0,1,2\}}) = L(a^*|a^*b(\varepsilon|aa^*b)^*aaa^*).$$

So $a^*|a^*b(\varepsilon|aa^*b)^*aaa^*$ is a regular expression whose matching strings comprise the language accepted by the NFA on Slide 25. (Clearly, one could simplify this to a smaller, but equivalent regular expression (in the sense of Slide 8), but we do not bother to do so.)

4.3 Complement and intersection of regular languages

We saw in Section 3.2 that part (a) of Kleene's Theorem allows us to answer question (a) on Slide 9. Now that we have proved the other half of the theorem, we can say more about question (b) on that slide.

Complementation Recall that on page 8 we mentioned that for each regular expression r over an alphabet Σ, we can find a regular expression $\sim(r)$ that determines the complement of the language determined by r:

$$L(\sim(r)) = \{u \in \Sigma^* \mid u \notin L(r)\}.$$

As we now show, this is a consequence of Kleene's Theorem.

$Not(M)$

- $States_{Not(M)} \stackrel{\text{def}}{=} States_M$
- $\Sigma_{Not(M)} \stackrel{\text{def}}{=} \Sigma_M$
- transitions of $Not(M)$ = transitions of M
- start state of $Not(M)$ = start state of M
- $Accept_{Not(M)} = \{q \in States_M \mid q \notin Accept_M\}$.

Provided M is a *deterministic* finite automaton, then u is accepted by $Not(M)$ iff it is not accepted by M:

$$L(Not(M)) = \{u \in \Sigma^* \mid u \notin L(M)\}.$$

Slide 26

Lemma 4.3.1. *If L is a regular language over alphabet Σ, then its complement $\{u \in \Sigma^* \mid u \notin L\}$ is also regular.*

Proof. Since L is regular, by definition there is a DFA M such that $L = L(M)$. Let $Not(M)$ be the DFA constructed from M as indicated on Slide 26. Then $\{u \in \Sigma^* \mid u \notin L\}$ is the set of strings accepted by $Not(M)$ and hence is regular. □

Given a regular expression r, by part (a) of Kleene's Theorem there is a DFA M such that $L(r) = L(M)$. Then by part (b) of the theorem applied to the DFA $Not(M)$, we can find a regular expression $\sim(r)$ so that $L(\sim(r)) = L(Not(M))$. Since

$$L(Not(M)) = \{u \in \Sigma^* \mid u \notin L(M)\} = \{u \in \Sigma^* \mid u \notin L(r)\},$$

this $\sim(r)$ is the regular expression we need for the complement of r.

Note. The construction given on Slide 26 can be applied to a finite automaton M whether or not it is deterministic. However, for $L(Not(M))$ to equal $\{u \in \Sigma^* \mid u \notin L(M)\}$ we need M to be deterministic. See Exercise 4.4.2.

Intersection As another example of the power of Kleene's Theorem, given regular expressions r_1 and r_2 we can show the existence of a regular expression $(r_1 \& r_2)$ with the property:

u matches $(r_1 \& r_2)$ iff u matches r_1 *and* u matches r_2.

This can be deduced from the following lemma.

Lemma 4.3.2. *If L_1 and L_2 are a regular languages over an alphabet Σ, then their intersection*

$$L_1 \cap L_2 \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid u \in L_1 \textit{ and } u \in L_2\}$$

is also regular.

Proof. Since L_1 and L_2 are regular languages, there are DFA M_1 and M_2 such that $L_i = L(M_i)$ $(i = 1, 2)$. Let $And(M_1, M_2)$ be the DFA constructed from M_1 and M_2 as on Slide 27. It is not hard to see that $And(M_1, M_2)$ has the property that any $u \in \Sigma^*$ is accepted by $And(M_1, M_2)$ iff it is accepted by both M_1 and M_2. Thus $L_1 \cap L_2 = L(And(M_1, M_2))$ is a regular language. □

$And(M_1, M_2)$

- states of $And(M_1, M_2)$ are all ordered pairs (q_1, q_2) with $q_1 \in States_{M_1}$ and $q_2 \in States_{M_2}$
- alphabet of $And(M_1, M_2)$ is the common alphabet of M_1 and M_2
- $(q_1, q_2) \xrightarrow{a} (q'_1, q'_2)$ in $And(M_1, M_2)$ iff $q_1 \xrightarrow{a} q'_1$ in M_1 and $q_2 \xrightarrow{a} q'_2$ in M_2
- start state of $And(M_1, M_2)$ is (s_{M_1}, s_{M_2})
- (q_1, q_2) accepting in $And(M_1, M_2)$ iff q_1 accepting in M_1 and q_2 accepting in M_2.

Slide 27

Thus given regular expressions r_1 and r_2, by part (a) of Kleene's Theorem we can find DFA M_1 and M_2 with $L(r_i) = L(M_i)$ $(i = 1, 2)$. Then by part (b) of the theorem we can find a regular expression $r_1 \& r_2$ so that $L(r_1 \& r_2) = L(And(M_1, M_2))$. Thus u matches $r_1 \& r_2$ iff $And(M_1, M_2)$ accepts u, iff both M_1 and M_2 accept u, iff u matches both r_1 and r_2, as required.

4.4 Exercises

Exercise 4.4.1. Use the construction in Section 4.1 to find a regular expression for the DFA M whose state set is $\{0, 1, 2\}$, whose start state is 0, whose only accepting state is 2, whose alphabet of input symbols is $\{a, b\}$, and whose next-state function is given by the following table.

$$\begin{array}{r|cc} \delta_M : & a & b \\ \hline 0 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 2 & 1 \end{array}$$

Exercise 4.4.2. The construction $M \mapsto Not(M)$ given on Slide 26 applies to both DFA and NFA; but for $L(Not(M))$ to be the complement of $L(M)$ we need M to be deterministic. Give an example of an alphabet Σ and a NFA M with set of input symbols Σ, such that $\{u \in \Sigma^* \mid u \notin L(M)\}$ is not the same set as $L(Not(M))$.

Exercise 4.4.3. Let $r = (a|b)^*ab(a|b)^*$. Find a complement for r over the alphabet $\Sigma = \{a, b\}$, i.e. a regular expressions $\sim(r)$ over the alphabet Σ satisfying $L(\sim(r)) = \{u \in \Sigma^* \mid u \notin L(r)\}$. Do the same for the alphabet $\{a, b, c\}$.

Tripos questions 1995.2.20 1994.3.3 1988.2.3 2000.2.7

5 The Pumping Lemma

In the context of programming languages, a typical example of a regular language (Slide 19) is the set of all strings of characters which are well-formed *tokens* (basic keywords, identifiers, etc) in a particular programming language, Java say. By contrast, the set of all strings which represent well-formed Java *programs* is a typical example of a language that is not regular. Slide 28 gives some simpler examples of non-regular languages. For example, there is no way to use a search based on matching a regular expression to find all the palindromes in a piece of text (although of course there are other kinds of algorithm for doing this).

Examples of non-regular languages

- The set of strings over $\{(,), a, b, \ldots, z\}$ in which the parentheses '(' and ')' occur well-nested.
- The set of strings over $\{a, b, \ldots, z\}$ which are *palindromes*, i.e. which read the same backwards as forwards.
- $\{a^n b^n \mid n \geq 0\}$

Slide 28

The intuitive reason why the languages listed on Slide 28 are not regular is that a machine for recognising whether or not any given string is in the language would need *infinitely* many different states (whereas a characteristic feature of the machines we have been using is that they have only *finitely* many states). For example, to recognise that a string is of the form $a^n b^n$ one would need to remember how many as had been seen before the first b is encountered, requiring countably many states of the form 'just_ seen_n_as'. This section make this intuitive argument rigorous and describes a useful way of showing that languages such as these are not regular.

The fact that a finite automaton does only have finitely many states means that as we look at longer and longer strings that it accepts, we see a certain kind of repetition—the ***pumping lemma property*** given on Slide 29.

The Pumping Lemma

For every regular language L, there is a number $\ell \geq 1$ satisfying the ***pumping lemma property***:

all $w \in L$ with $length(w) \geq \ell$ can be expressed as a concatenation of three strings, $w = u_1 v u_2$, where u_1, v and u_2 satisfy:

- $length(v) \geq 1$
  (i.e. $v \neq \varepsilon$)
- $length(u_1 v) \leq \ell$
- for all $n \geq 0$, $u_1 v^n u_2 \in L$
  (i.e. $u_1 u_2 \in L$, $\quad u_1 v u_2 \in L$ [but we knew that anyway],
  $u_1 v v u_2 \in L$, $\quad u_1 v v v u_2 \in L$, $\quad$ etc).

Slide 29

5.1 Proving the Pumping Lemma

Since L is regular, it is equal to the set $L(M)$ of strings accepted by some DFA M. Then *we can take the number ℓ mentioned on Slide 29 to be the number of states in M.* For suppose $w = a_1 a_2 \dots a_n$ with $n \geq \ell$. If $w \in L(M)$, then there is a transition sequence as shown at the top of Slide 30. Then w can be split into three pieces as shown on that slide. Note that by choice of i and j, $length(v) = j - i \geq 1$ and $length(u_1 v) = j \leq \ell$. So it just remains to check that $u_1 v^n u_2 \in L$ for all $n \geq 0$. As shown on the lower half of Slide 30, the string v takes the machine M from state q_i back to the same state (since $q_i = q_j$). So for any n, $u_1 v^n u_2$ takes us from the initial state $s_M = q_o$ to q_i, then n times round the loop from q_i to itself, and then from q_i to $q_n \in Accept_M$. Therefore for any $n \geq 0$, $u_1 v^n u_2$ is accepted by M, i.e. $u_1 v^n u_2 \in L$. $\square$

Note. In the above construction it is perfectly possible that $i = 0$, in which case u_1 is the null-string, ε.

If $n \geq \ell =$ number of states of M, then in

$$s_M = \underbrace{q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_\ell} q_\ell}_{\ell+1 \text{ states}} \cdots \xrightarrow{a_n} q_n \in Accept_M$$

$q_0, \ldots, q_\ell$ can't all be distinct states. So $q_i = q_j$ for some $0 \leq i < j \leq \ell$. So the above transition sequence looks like

$$\begin{tikzcd} s_M = q_0 \arrow[r, "u_1"{above}, "*"{near end}] & q_i \arrow[r, phantom, "="] \arrow[r, bend left=60, "v"] & q_j^{*} \arrow[r, "u_2"{above}, "*"{near end}] & q_n \in Accept_M \end{tikzcd}$$

where

$$u_1 \stackrel{\text{def}}{=} a_1 \ldots a_i \qquad v \stackrel{\text{def}}{=} a_{i+1} \ldots a_j \qquad u_2 \stackrel{\text{def}}{=} a_{j+1} \ldots a_n.$$

Slide 30

Remark 5.1.1. One consequence of the pumping lemma property of L and ℓ is that if there is any string w in L of length $\geq \ell$, then L contains arbitrarily long strings. (We just 'pump up' w by increasing n.)

If you did Exercise 3.3.3, you will know that if L is a *finite* set of strings then it is regular. In this case, what is the number ℓ with the property on Slide 29? The answer is that we can take any ℓ strictly greater than the length of any string in the finite set L. Then the Pumping Lemma property is trivially satisfied because there are no $w \in L$ with $length(w) \geq \ell$ for which we have to check the condition!

5.2 Using the Pumping Lemma

The Pumping Lemma (Slide 5.1) says that every regular language has a certain property—namely that there exists a number ℓ with the pumping lemma property. So to show that a language L is *not* regular, it suffices to show that no $\ell \geq 0$ possesses the pumping lemma property for the language L. Because the pumping lemma property involves quite a complicated alternation of quantifiers, it will help to spell out explicitly what is its negation. This is done on Slide 31. Slide 32 gives some examples.

How to use the Pumping Lemma to prove that a language L is *not* regular

For each $\ell \geq 1$, find some $w \in L$ of length $\geq \ell$ so that

$$(\dagger) \quad \left\{ \begin{array}{l} \text{no matter how } w \text{ is split into three, } w = u_1 v u_2, \\ \text{with } length(u_1 v) \leq \ell \text{ and } length(v) \geq 1, \\ \text{there is some } n \geq 0 \text{ for which } u_1 v^n u_2 \text{ is } \textit{not} \text{ in } L. \end{array} \right.$$

Slide 31

Examples

(i) $L_1 \stackrel{\text{def}}{=} \{a^n b^n \mid n \geq 0\}$ *is not regular.*

[For each $\ell \geq 1$, $a^\ell b^\ell \in L_1$ is of length $\geq \ell$ and has property (†) on Slide 31.]

(ii) $L_2 \stackrel{\text{def}}{=} \{w \in \{a,b\}^* \mid w$ *a palindrome*$\}$ *is not regular.*

[For each $\ell \geq 1$, $a^\ell b a^\ell \in L_1$ is of length $\geq \ell$ and has property (†).]

(iii) $L_3 \stackrel{\text{def}}{=} \{a^p \mid p$ *prime*$\}$ *is not regular.*

[For each $\ell \geq 1$, we can find a prime p with $p > 2\ell$ and then $a^p \in L_3$ has length $\geq \ell$ and has property (†).]

Slide 32

Proof of the examples on Slide 32. We use the method on Slide 31.

(i) For any $\ell \geq 1$, consider the string $w = a^\ell b^\ell$. It is in L_1 and has length $\geq \ell$. We show that property (†) holds for this w. For suppose $w = a^\ell b^\ell$ is split as $w = u_1 v u_2$ with $\mathit{length}(u_1 v) \leq \ell$ and $\mathit{length}(v) \geq 1$. Then $u_1 v$ must consist entirely of as, so $u_1 = a^r$ and $v = a^s$ say, and hence $u_2 = a^{\ell-r-s} b^\ell$. Then the case $n = 0$ of $u_1 v^n u_2$ is not in L_1 since

$$u_1 v^0 u_2 = u_1 u_2 = a^r (a^{\ell-r-s} b^\ell) = a^{\ell-s} b^\ell$$

and $a^{\ell-s} b^\ell \notin L_1$ because $\ell - s \neq \ell$ (since $s = \mathit{length}(v) \geq 1$).

(ii) The argument is very similar to that for example (i), but starting with the palindrome $w = a^\ell b a^\ell$. Once again, the $n = 0$ of $u_1 v^n u_2$ yields a string $u_1 u_2 = a^{\ell-s} b a^\ell$ which is not a palindrome (because $\ell - s \neq \ell$).

(iii) Given $\ell \geq 1$, since there are infinitely many primes p, we can certainly find one satisfying $p > 2\ell$. I claim that $w = a^p$ has property (†). For suppose $w = a^p$ is split as $w = u_1 v u_2$ with $\mathit{length}(u_1 v) \leq \ell$ and $\mathit{length}(v) \geq 1$. Letting $r \stackrel{\text{def}}{=} \mathit{length}(u_1)$ and $s \stackrel{\text{def}}{=} \mathit{length}(v)$, so that $\mathit{length}(u_2) = p - r - s$, we have

$$u_1 v^{p-s} u_2 = a^r a^{s(p-s)} a^{p-r-s} = a^{sp-s^2+p-s} = a^{(s+1)(p-s)}.$$

Now $(s+1)(p-s)$ is not prime, because $s + 1 > 1$ (since $s = \mathit{length}(v) \geq 1$) and $p - s > 2\ell - \ell = \ell \geq 1$ (since $p > 2\ell$ by choice, and $s \leq r + s = \mathit{length}(u_1 v) \leq \ell$). Therefore $u_1 v^n u_2 \notin L_3$ when $n = p - s$.

□

Remark 5.2.1. Unfortunately, the method on Slide 31 can't cope with every non-regular language. This is because the pumping lemma property is a necessary, but not a sufficient condition for a language to be regular. In other words there do exist languages L for which a number $\ell \geq 1$ can be found satisfying the pumping lemma property on Slide 29, but which nonetheless, are not regular. Slide 33 gives an example of such an L.

Example of a non-regular language that satisfies the 'pumping lemma property'

$$L \stackrel{\text{def}}{=} \{c^m a^n b^n \mid m \geq 1 \text{ and } n \geq 0\} \cup \{a^m b^n \mid m, n \geq 0\}$$

satisfies the pumping lemma property on Slide 29 with $\ell = 1$.

[For any $w \in L$ of length ≥ 1, can take $u_1 = \varepsilon$, $v =$ first letter of w, $u_2 =$ rest of w.]

But L is not regular. [See Exercise 5.4.2.]

Slide 33

5.3 Decidability of language equivalence

The proof of the Pumping Lemma provides us with a positive answer to question (c) on Slide 9. In other words, it provides a method that, given any two regular expressions r_1 and r_2 (over the same alphabet Σ) decides whether or not the languages they determine are equal, $L(r_1) = L(r_2)$.

First note that this problem can be reduced to *deciding whether or not the set of strings accepted by any given DFA is empty.* For $L(r_1) = L(r_2)$ iff $L(r_1) \subseteq L(r_2)$ and $L(r_2) \subseteq L(r_1)$. Using the results about complementation and intersection in Section 4.3, we can reduce the question of whether or not $L(r_1) \subseteq L(r_2)$ to the question of whether or not $L(r_1\&(\sim r_2)) = \emptyset$, since

$$L(r_1) \subseteq L(r_2) \quad \text{iff} \quad L(r_1) \cap \{u \in \Sigma^* \mid u \notin L(r_2)\} = \emptyset.$$

By Kleene's theorem, given r_1 and r_2 we can first construct regular expressions $r_1\&(\sim r_2)$ and $r_2\&(\sim r_1)$, then construct DFAs M_1 and M_2 such that $L(M_1) = L(r_1\&(\sim r_2))$ and $L(M_2) = L(r_2\&(\sim r_1))$. Then r_1 and r_2 are equivalent iff the languages accepted by M_1 and by M_2 are both empty.

The fact that, given any DFA M, one can decide whether or not $L(M) = \emptyset$ follows from the Lemma on Slide 34. For then, to check whether or not $L(M)$ is empty, we just have to check whether or not any of the finitely many strings of length less than the number of states of M is accepted by M.

Lemma *If a DFA M accepts any string at all, it accepts one whose length is less than the number of states in M.*

Proof. Suppose M has ℓ states (so $\ell \geq 1$). If $L(M)$ is not empty, then we can find an element of it *of shortest length*, $a_1 a_2 \ldots a_n$ say (where $n \geq 0$). Thus there is a transition sequence

$$s_M = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n \in Accept_M.$$

If $n \geq \ell$, then not all the $n+1$ states in this sequence can be distinct and we can shorten it as on Slide 30. But then we would obtain a strictly shorter string in $L(M)$ contradicting the choice of $a_1 a_2 \ldots a_n$. So we must have $n < \ell$. $\square$

Slide 34

5.4 Exercises

Exercise 5.4.1. Show that the first language mentioned on Slide 28 is not regular.

Exercise 5.4.2. Show that there is no DFA M for which $L(M)$ is the language on Slide 33. [Hint: argue by contradiction. If there were such an M, consider the DFA M' with the same states as M, with alphabet of input symbols just consisting of a and b, with transitions all those of M which are labelled by a or b, with start state $\delta_M(s_M, c)$ (where s_M is the start state of M), and with the same accepting states as M. Show that the language accepted by M' has to be $\{a^n b^n \mid n \geq 0\}$ and deduce that no such M can exist.]

Exercise 5.4.3. Check the claim made on Slide 33 that the language mentioned there satisfies the pumping lemma property of Slide 29 with $\ell = 1$.

Tripos questions 2002.2.9 2001.2.7 1999.2.7 1998.2.7 1996.2.1(j) 1996.2.8 1995.2.27 1993.6.12 1991.4.6 1989.6.12

6 Grammars

We have seen that regular languages can be specified in terms of finite automata that accept or reject strings, and equivalently, in terms of patterns, or regular expressions, which strings are to match. This section briefly introduces an alternative, 'generative' way of specifying languages.

6.1 Context-free grammars

Some production rules for 'English' sentences

$$
\begin{aligned}
\mathtt{SENTENCE} &\to \mathtt{SUBJECT}\ \mathtt{VERB}\ \mathtt{OBJECT}\\
\mathtt{SUBJECT} &\to \mathtt{ARTICLE}\ \mathtt{NOUNPHRASE}\\
\mathtt{OBJECT} &\to \mathtt{ARTICLE}\ \mathtt{NOUNPHRASE}\\
\mathtt{ARTICLE} &\to \mathtt{a}\\
\mathtt{ARTICLE} &\to \mathtt{the}\\
\mathtt{NOUNPHRASE} &\to \mathtt{NOUN}\\
\mathtt{NOUNPHRASE} &\to \mathtt{ADJECTIVE}\ \mathtt{NOUN}\\
\mathtt{ADJECTIVE} &\to \mathtt{big}\\
\mathtt{ADJECTIVE} &\to \mathtt{small}\\
\mathtt{NOUN} &\to \mathtt{cat}\\
\mathtt{NOUN} &\to \mathtt{dog}\\
\mathtt{VERB} &\to \mathtt{eats}
\end{aligned}
$$

Slide 35

Slide 35 gives an example of a context-free grammar for generating strings over the seven element alphabet

$$\Sigma \stackrel{\text{def}}{=} \{\mathtt{a}, \mathtt{big}, \mathtt{cat}, \mathtt{dog}, \mathtt{eats}, \mathtt{small}, \mathtt{the}\}.$$

The elements of the alphabet are called ***terminals*** for reasons that will emerge below. The grammar uses finitely many extra symbols, called ***non-terminals***, namely the eight symbols

$$\mathtt{ADJECTIVE}, \mathtt{ARTICLE}, \mathtt{NOUN}, \mathtt{NOUNPHRASE}, \mathtt{OBJECT}, \mathtt{SENTENCE}, \mathtt{SUBJECT}, \mathtt{VERB}.$$

One of these is designated as the ***start symbol***. In this case it is SENTENCE (because we are interested in generating sentences). Finally, the context-free grammar contains a finite set of ***production*** rules, each of which consists of a pair, written $x \to u$, where x is one of the non-terminals and u is a string of terminals and non-terminals. In this case there are twelve productions, as shown on the slide.

The idea is that we begin with the start symbol SENTENCE and use the productions to continually replace non-terminal symbols by strings. At successive stages in this process we have a string which may contain both terminals and non-terminals. We choose one of the non-terminals in the string and a production which has that non-terminal as its left-hand side. Replacing the non-terminal by the right-hand side of the production we obtain the next string in the sequence, or ***derivation*** as it is called. The derivation stops when we obtain a string containing only terminals. The set of strings over Σ that may be obtained in this way from the start symbol is by definition the ***language generated the context-free grammar***.

A derivation

$$
\begin{aligned}
\underline{\texttt{SENTENCE}} &\to \underline{\texttt{SUBJECT}}\ \texttt{VERB OBJECT} \\
&\to \underline{\texttt{ARTICLE}}\ \texttt{NOUNPHRASE VERB OBJECT} \\
&\to \texttt{the NOUNPHRASE}\ \underline{\texttt{VERB}}\ \texttt{OBJECT} \\
&\to \texttt{the}\ \underline{\texttt{NOUNPHRASE}}\ \texttt{eats OBJECT} \\
&\to \texttt{the}\ \underline{\texttt{ADJECTIVE}}\ \texttt{NOUN eats OBJECT} \\
&\to \texttt{the big}\ \underline{\texttt{NOUN}}\ \texttt{eats OBJECT} \\
&\to \texttt{the big cat eats}\ \underline{\texttt{OBJECT}} \\
&\to \texttt{the big cat eats}\ \underline{\texttt{ARTICLE}}\ \texttt{NOUNPHRASE} \\
&\to \texttt{the big cat eats a}\ \underline{\texttt{NOUNPHRASE}} \\
&\to \texttt{the big cat eats a}\ \underline{\texttt{ADJECTIVE}}\ \texttt{NOUN} \\
&\to \texttt{the big cat eats a small}\ \underline{\texttt{NOUN}} \\
&\to \texttt{the big cat eats a small dog}
\end{aligned}
$$

Slide 36

For example, the string

`the big cat eats a small dog`

is in this language, as witnessed by the derivation on Slide 36, in which we have indicated left-hand sides of production rules by underlining. On the other hand, the string

(4) `the dog a`

is *not* in the language, because there is no derivation from SENTENCE to the string. (Why?)

Remark 6.1.1. The phrase 'context-free' refers to the fact that in a derivation we are allowed to replace an occurrence of a non-terminal by the right-hand side of a production without regard to the strings that occur on either side of the occurrence (its 'context'). A more general form of grammar (a 'type 0 grammar' in the Chomsky hierarchy—see page 257 of Kozen's book, for example) has productions of the form $u \to v$ where u and v are arbitrary strings of terminals and non-terminals. For example a production of the form

$$\texttt{a ADJECTIVE cat} \to \texttt{dog}$$

would allow occurrences of '`ADJECTIVE`' that occur between '`a`' and '`cat`' to be replaced by '`dog`', deleting the surrounding symbols at the same time. This kind of production is not permitted in a context-free grammar.

Example of Backus-Naur Form (BNF)

Terminals:

$$\texttt{x} \quad ' \quad + \quad - \quad * \quad (\quad)$$

Non-terminals:

$$\texttt{id} \quad \texttt{op} \quad \texttt{exp}$$

Start symbol:

$$\texttt{exp}$$

Productions:

$$\begin{aligned} \texttt{id} &::= \texttt{x} \mid \texttt{id}' \\ \texttt{op} &::= + \mid - \mid * \\ \texttt{exp} &::= \texttt{id} \mid \texttt{exp op exp} \mid (\texttt{exp}) \end{aligned}$$

Slide 37

6.2 Backus-Naur Form

It is quite likely that the same non-terminal will appear on the left-hand side of several productions in a context-free grammar. Because of this, it is common to use a more compact notation for specifying productions, called ***Backus-Naur Form*** (BNF), in which all the productions for a given non-terminal are specified together, with the different right-hand sides being separated by the symbol '$|$'. BNF also tends to use the symbol '$::=$' rather than '$\to$' in the notation for productions. An example of a context-free grammar in BNF is given on Slide 37. Written out in full, the context-free grammar on this slide has eight productions, namely:

$$\begin{aligned}
\texttt{id} &\to \texttt{x}\\
\texttt{id} &\to \texttt{id}'\\
\texttt{op} &\to +\\
\texttt{op} &\to -\\
\texttt{op} &\to *\\
\texttt{exp} &\to \texttt{id}\\
\texttt{exp} &\to \texttt{exp}\,\texttt{op}\,\texttt{exp}\\
\texttt{exp} &\to (\texttt{exp})
\end{aligned}$$

The language generated by this grammar is supposed to represent certain arithmetic expressions. For example

(5) $$\texttt{x} + (\texttt{x}'')$$

is in the language, but

(6) $$\texttt{x} + (\texttt{x})''$$

is not. (See Exercise 6.4.2.)

A context-free grammar for the language
$\{a^n b^n \mid n \geq 0\}$

Terminals:

$$a \qquad b$$

Non-terminal:

$$I$$

Start symbol:

$$I$$

Productions:

$$I ::= \varepsilon \mid aIb$$

Slide 38

6.3 Regular grammars

A language L over an alphabet Σ is ***context-free*** iff L is the set of strings generated by some context-free grammar (with set of terminals Σ). The context-free grammar on Slide 38 generates the language $\{a^n b^n \mid n \geq 0\}$. We saw in Section 5.2 that this is not a regular language. So the class of context-free languages is not the same as the class of regular languages. Nevertheless, as Slide 39 points out, every regular language is context-free. For the grammar defined on that slide clearly has the property that derivations from the start symbol to a string in Σ^* must be of the form of a finite number of productions of the first kind followed by a single production of the second kind, i.e.

$$s_M \to a_1 q_1 \to a_1 a_2 q_2 \to \cdots \to a_1 a_2 \dots a_n q_n \to a_1 a_2 \dots a_n$$

where in M the following transition sequence holds

$$s_M \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n \in Accept_M.$$

Thus a string is in the language generated by the grammar iff it is accepted by M.

Every regular language is context-free

Given a DFA M, the set $L(M)$ of strings accepted by M can be generated by the following context-free grammar:

set of terminals = Σ_M

set of non-terminals = $States_M$

start symbol = start state of M

productions of two kinds:

$q \to aq'$	whenever $q \xrightarrow{a} q'$ in M
$q \to \varepsilon$	whenever $q \in Accept_M$

Slide 39

Definition A context-free grammar is ***regular*** iff all its productions are of the form

$$x \to uy$$

or

$$x \to u$$

where u is a string of terminals and x and y are non-terminals.

Theorem

(a) *Every language generated by a regular grammar is a regular language (i.e. is the set of strings accepted by some DFA).*

(b) *Every regular language can be generated by a regular grammar.*

Slide 40

It is possible to single out context-free grammars of a special form, called ***regular*** (or ***right linear***), which do generate regular languages. The definition is on Slide 40. Indeed, as the theorem on that slide states, this type of grammar generates *all* possible regular languages.

Proof of the Theorem on Slide 40. First note that part (b) of the theorem has already been proved, because the context-free grammar generating $L(M)$ on Slide 39 is a regular grammar (of a special kind).

To prove part (a), given a regular grammar we have to construct a DFA M whose set of accepted strings coincides with the strings generated by the grammar. By the Subset Construction (Theorem on Slide 18), it is enough to construct an NFA$^{\varepsilon}$ with this property. This makes the task much easier. We take the states of M to be the non-terminals, augmented by some extra states described below. Of course the alphabet of input symbols of M should be the set of terminal symbols of the grammar. The start state is the start symbol. Finally, the transitions and the accepting states of M are defined as follows.

(i) For each production of the form $q \to uq'$ with $length(u) \geq 1$, say $u = a_1 a_2 \ldots a_n$ with $n \geq 1$, we add $n - 1$ fresh states $q_1, q_2, q_3, \ldots, q_{n-1}$ to the automaton and transitions

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_3 \xrightarrow{a_3} \cdots q_{n-1} \xrightarrow{a_n} q'.$$

(ii) For each production of the form $q \to uq'$ with $length(u) = 0$, i.e. with $u = \varepsilon$, we add an ε-transition

$$q \xrightarrow{\varepsilon} q'.$$

(iii) For each production of the form $q \to u$ with $length(u) \geq 1$, say $u = a_1 a_2 \dots a_n$ with $n \geq 1$, we add n fresh states $q_1, q_2, q_3, \dots, q_n$ to the automaton and transitions

$$q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots \xrightarrow{a_n} q_n.$$

Moreover we make the state q_n accepting.

(iv) For each production of the form $q \to u$ with $length(u) = 0$, i.e. with $u = \varepsilon$, we do not add in any new states or transitions, but we do make q an accepting state.

If we have a transition sequence in M of the form $s_M \stackrel{u}{\Rightarrow} q$ with $q \in Accept_M$, we can divide it up into pieces according to where non-terminals occur and then convert each piece into a use of one of the production rules, thereby forming a derivation of u in the grammar. Reversing this process, every derivation of a string of terminals can be converted into a transition sequence in the automaton from the start state to an accepting state. Thus this NFA$^\varepsilon$does indeed accept exactly the set of strings generated by the given regular grammar. □

6.4 Exercises

Exercise 6.4.1. Why is the string (4) not in the language generated by the context-free grammar in Section 6.1?

Exercise 6.4.2. Give a derivation showing that (5) is in the language generated by the context-free grammar on Slide 37. Prove that (6) is not in that language. [Hint: show that if u is a string of terminals and non-terminals occurring in a derivation of this grammar and that '$'$' occurs in u, then it does so in a substring of the form v', or v'', or v''', etc., where v is either `x` or `id`.]

Exercise 6.4.3. Give a context-free grammar generating all the palindromes over the alphabet $\{a, b\}$ (cf. Slide 28).

Exercise 6.4.4. Give a context-free grammar generating all the regular expressions over the alphabet $\{a, b\}$.

Exercise 6.4.5. Using the construction given in the proof of part (a) of the Theorem on Slide 40, convert the regular grammar with start symbol q_0 and productions

$$\begin{aligned} &q_0 \to \varepsilon \\ &q_0 \to abq_0 \\ &q_0 \to cq_1 \\ &q_1 \to ab \end{aligned}$$

into an NFA$^\varepsilon$whose language is that generated by the grammar.

Exercise 6.4.6. Is the language generated by the context-free grammar on Slide 35 a regular language? What about the one on Slide 37?

Tripos questions 2002.2.1(d) 1997.2.7 1996.2.1(k) 1994.4.3 1991.3.8 1990.6.10