Formal Languages and Automata

5 lectures for

University of Cambridge
2019–2020 Computer Science Tripos
Part IA Discrete Mathematics
by Prof. Frank Stajano

Originally written by **Prof. Andrew Pitts**

Minor tweaks by Prof. Ian Leslie and Prof. Frank Stajano

Revision 12 of 2020-01-27 03:16:35 +0000 (Mon, 27 Jan 2020)

Contents and Syllabus

Common theme: mathematical techniques for defining formal languages and reasoning about their properties.

Key concepts: inductive definitions, automata

Relevant to:

Part IB	Compiler Construction, Computation Theory, Complexity Theory, Semantics of Programming Languages
Part II	Natural Language Processing, Optimising Compilers, Denotational Semantics, Hoare Logic and Model Checking

Formal Languages

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:

- $\{0,1,2,3,4,5,6,7,8,9\}$, **10**-element set of decimal digits.
- $\{a,b,c,\ldots,x,y,z\}$, **26**-element set of lower-case characters of the English language.
- $\{S \mid S \subseteq \{0,1,2,3,4,5,6,7,8,9\}\}$, $\mathbf{2^{10}}$-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.

Strings over an alphabet

A **string of length** n (for $n = 0, 1, 2, \ldots$) over an alphabet Σ is just an ordered n-tuple of elements of Σ, written without punctuation.

Σ^* denotes the set of all strings over Σ of any finite length.

Examples:

notation for the string of length 0

- If $\Sigma = \{a, b, c\}$, then ε, a, ab, aac, and $bbac$ are strings over Σ of lengths zero, one, two, three and four respectively.
- If $\Sigma = \{a\}$, then Σ^* contains ε, a, aa, aaa, $aaaa$, etc.
- If $\Sigma = \emptyset$ (the empty set), then $\Sigma^* = \{\varepsilon\}$.

Notes

- There is a unique string of length zero over Σ, called the **null string** (or **empty string**) and denoted ε, no matter which alphabet Σ we are talking about.
- We make no notational distinction between a symbol $a \in \Sigma$ and the string of length 1 containing a. Thus we regard Σ as a subset of Σ^*.
- $\emptyset$, $\{\varepsilon\}$ and ε are three different things!
  - $\emptyset$ is the (unique) set with no elements,
  - $\{\varepsilon\}$ is a set with one element (the null string),
  - ε is the string of length 0.
- The length of a string $u \in \Sigma^*$ is denoted $|u|$.
- We are not concerned here with data structures and algorithms for implementing strings (so strings and finite lists are interchangeable concepts here).
- **Warning!** the symbol * is highly overloaded – it means different things in different contexts in this course. (The same comment applies to the symbol ε and, to a lesser extent, the symbol $\emptyset$.)

Concatenation of strings

The **concatenation** of two strings u and v is the string uv obtained by joining the strings end-to-end. This generalises to the concatenation of three or more strings.

Examples:

If $\Sigma = \{a, b, c, \ldots, z\}$ and $u, v, w \in \Sigma^*$ are $u = ab$, $v = ra$ and $w = cad$, then

$$vu = raab$$
$$uu = abab$$
$$wv = cadra$$
$$uvwuv = abracadabra$$

Notes

- Concatenation satisfies:

$$u\varepsilon = u = \varepsilon u$$
$$(uv)w = uvw = u(vw)$$
(but in general $uv \neq vu$)

- $|uv| = |u| + |v|$

Notation

If $u \in \Sigma^*$, then u^n denotes n copies of u concatenated together. By convention $u^0 = \varepsilon$.

Formal languages

An extensional view of what constitutes a formal language is that it is completely determined by the set of 'words in the dictionary':

Given an alphabet Σ, we call any subset of Σ^* a (formal) **language** over the alphabet Σ.

We will use inductive definitions to describe languages in terms of grammatical rules for generating subsets of Σ^*.

Inductive Definitions

Axioms and rules

for inductively defining a subset of a given set U

- **axioms** $\boxed{\dfrac{}{a}}$ are specified by giving an element a of U
- **rules** $\boxed{\dfrac{h_1 \; h_2 \; \cdots \; h_n}{c}}$

  are specified by giving a finite subset $\{h_1, h_2, \ldots, h_n\}$ of U (the **hypotheses** of the rule) and an element c of U (the **conclusion** of the rule)

Derivations

Given a set of axioms and rules for inductively defining a subset of a given set U, a **derivation** (or proof) that a particular element $u \in U$ is in the subset is by definition

> a finite tree with vertices labelled by elements of U and such that:
>
> - the root of the tree is u (the conclusion of the whole derivation),
> - each vertex of the tree is the conclusion of a rule whose hypotheses are the children of the node,
> - each leaf of the tree is an axiom.

Example

$U = \{a, b\}^*$

axiom: $\dfrac{}{\varepsilon}$

rules: $\dfrac{u}{aub}$ $\dfrac{u}{bua}$ $\dfrac{u \quad v}{uv}$ (for all $u, v \in U$)

Example derivations:

$$\dfrac{\dfrac{\varepsilon}{ab} \quad \dfrac{\dfrac{\varepsilon}{ab}}{aabb}}{abaabb} \qquad \dfrac{\dfrac{\varepsilon}{ba} \quad \dfrac{\varepsilon}{ab}}{baab}}{abaabb}$$

Inductively defined subsets

Given a set of axioms and rules over a set U, the subset of U **inductively defined** by the axioms and rules consists of all and only the elements $u \in U$ for which there is a derivation with conclusion u.

For example, for the axioms and rules on Slide 14

- $abaabb$ is in the subset they inductively define (as witnessed by either derivation on that slide)
- $abaab$ is not in that subset (there is no derivation with that conclusion – why?)

(In fact $u \in \{a,b\}^*$ is in the subset iff it contains the same number of a and b symbols.)

Notes

- Axioms are special cases of rules – the ones where $n = 0$, i.e. the set of hypotheses is empty.
- We are generally interested in inductive definitions of subsets that are infinite. An inductive definition with only finitely many axioms and rules defines a finite subset. (Why?) So we usually have to consider infinite sets of axioms and rules. However, those sets are usually specified *schematically*: an axiom scheme, or a rule scheme is a template involving variables that can be instantiated to get a whole family of actual axioms or rules.

  For example, on Slide 14, we used the rule scheme $\dfrac{u}{aub}$ where u is meant to be instantiated with any string over the alphabet $\{a, b\}$. Thus this rule scheme stands for the infinite collection of rules $\dfrac{\varepsilon}{ab}$, $\dfrac{a}{aab}$, $\dfrac{b}{abb}$, $\dfrac{aa}{aaab}$, etc.
- It is sometimes convenient to flatten derivations into finite lists, because they are easier to fit on a page. The last element of the list is the conclusion of the derivation. Every element of the list is either an axiom, or the conclusion of a rule all of whose hypotheses occur earlier in the list.
- The fact that an element is in an inductively defined subset may be witnessed by more than one derivation (see the example on Slide 14).
- In general, there is no sure-fire, algorithmic method for showing that an element is *not* in a particular inductively defined subset.

Example: transitive closure

Given a binary relation $R \subseteq X \times X$ on a set X, its **transitive closure** R^+ is the smallest (for subset inclusion) binary relation on X which contains R and which is **transitive** ($\forall x, y, z \in X.\ (x, y) \in R^+ \ \&\ (y, z) \in R^+ \Rightarrow (x, z) \in R^+$).

R^+ is equal to the subset of $X \times X$ inductively defined by

axioms $\dfrac{}{(x, y)}$ (for all $(x, y) \in R$)

rules $\dfrac{(x, y) \qquad (y, z)}{(x, z)}$ (for all $x, y, z \in X$)

Example: reflexive-transitive closure

Given a binary relation $R \subseteq X \times X$ on a set X, its **reflexive-transitive closure** R^* is defined to be the smallest binary relation on X which contains R, is both transitive and **reflexive** ($\forall x \in X.\ (x,x) \in R^*$).

R^* is equal to the subset of $X \times X$ inductively defined by

axioms $\dfrac{}{(x,y)}$ (for all $(x,y) \in R$) $\qquad \dfrac{}{(x,x)}$ (for all $x \in X$)

rules $\dfrac{(x,y) \qquad (y,z)}{(x,z)}$ (for all $x,y,z \in X$)

we can use Rule Induction (Slide 19) to prove this

Rule Induction

Theorem. The subset $I \subseteq U$ inductively defined by a collection of axioms and rules is closed under them and is the least such subset: if $S \subseteq U$ is also closed under the axioms and rules, then $I \subseteq S$.

Given axioms and rules for inductively defining a subset of a set U, we say that a subset $S \subseteq U$ is **closed under the axioms and rules** if

- for every axiom $\dfrac{}{a}$, it is the case that $a \in S$
- for every rule $\dfrac{h_1 \; h_2 \; \cdots \; h_n}{c}$, if $h_1, h_2, \ldots, h_n \in S$, then $c \in S$.

Rule Induction

Theorem. The subset $I \subseteq U$ inductively defined by a collection of axioms and rules is closed under them and is the least such subset: if $S \subseteq U$ is also closed under the axioms and rules, then $I \subseteq S$.

We use a similar approach as method of proof: given a property $P(u)$ of elements of U, to prove $\forall u \in I.\, P(u)$ it suffices to show

- **base cases:** $P(a)$ holds for each axiom $\dfrac{}{a}$
- **induction steps:** $P(h_1) \,\&\, P(h_2) \,\&\, \cdots \,\&\, P(h_n) \Rightarrow P(c)$ holds for each rule $\dfrac{h_1 \; h_2 \; \cdots \; h_n}{c}$

(To see this, apply the theorem with $S = \{u \in U \mid P(u)\}$.)

Proof of the Theorem on Slide 19

I is closed under any of the axioms $\dfrac{}{a}$, because a is a derivation of length 1 showing that $a \in I$. I is closed under any of the rules $\dfrac{h_1 \ \cdots \ h_n}{c}$, because if each h_i is in I, there is a derivation D_i with conclusion h_i; and then $\dfrac{D_1 \ \cdots \ D_n}{c}$ is a derivation (why?) with conclusion $c \in I$.

Now suppose $S \subseteq U$ is some subset closed under the axioms and rules. We can use mathematical induction to prove

$$\forall n.\ \text{all derivations of height} \leq n \ \text{have their conclusion in } S \qquad (*)$$

Hence all derivations have their conclusions in S; and therefore $I \subseteq S$, as required. □

[Proof of $(*)$ by mathematical induction:

Base case $n = 0$: trivial, because there are no derivations of height 0.

Induction step for $n+1$: suppose D is a derivation of height $\leq n+1$, with conclusion c – say $D = \dfrac{D_1 \ \cdots \ D_m}{c}$ (some $m \geq 0$). We have to show $c \in S$. Note that each D_i is a derivation of height $\leq n$ and so by induction hypothesis its conclusion, c_i say, is in S. Since D is a well-formed derivation, $\dfrac{c_1 \ \cdots \ c_m}{c}$ has to be a rule (or $m = 0$ and it is an axiom). Since S is closed under the axioms and rules and each c_i is in S, we conclude that $c \in S$. □]

Example: reflexive-transitive closure

Given a binary relation $R \subseteq X \times X$ on a set X, its **reflexive-transitive closure** R^* is defined to be the smallest binary relation on X which contains R, is both transitive and **reflexive** ($\forall x \in X.\ (x,x) \in R^*$).

R^* is equal to the subset of $X \times X$ inductively defined by

axioms $\dfrac{}{(x,y)}$ (for all $(x,y) \in R$) $\qquad \dfrac{}{(x,x)}$ (for all $x \in X$)

rules $\dfrac{(x,y) \qquad (y,z)}{(x,z)}$ (for all $x,y,z \in X$)

we can use Rule Induction (Slide 19) to prove this, since $S \subseteq X \times X$ being closed under the axioms & rules is the same as it containing R, being reflexive and being transitive.

Example using rule induction

Let I be the subset of $\{a,b\}^*$ inductively defined by the axioms and rules on Slide 14.

For $u \in \{a,b\}^*$, let $P(u)$ be the property

u contains the same number of a and b symbols

We can prove $\forall u \in I.\, P(u)$ by rule induction:

- **base case:** $P(\varepsilon)$ is true (the number of as and bs is zero!)
- **induction steps:** if $P(u)$ and $P(v)$ hold, then clearly so do $P(aub)$, $P(bua)$ and $P(uv)$.

(It's not so easy to show $\forall u \in \{a,b\}^*.\, P(u) \Rightarrow u \in I$ – rule induction for I is not much help for that.)

Abstract Syntax Trees

Concrete syntax: strings of symbols

- possibly including symbols to disambiguate the semantics (brackets, white space, *etc*),
- or that have no semantic content (*e.g.* syntax for comments).

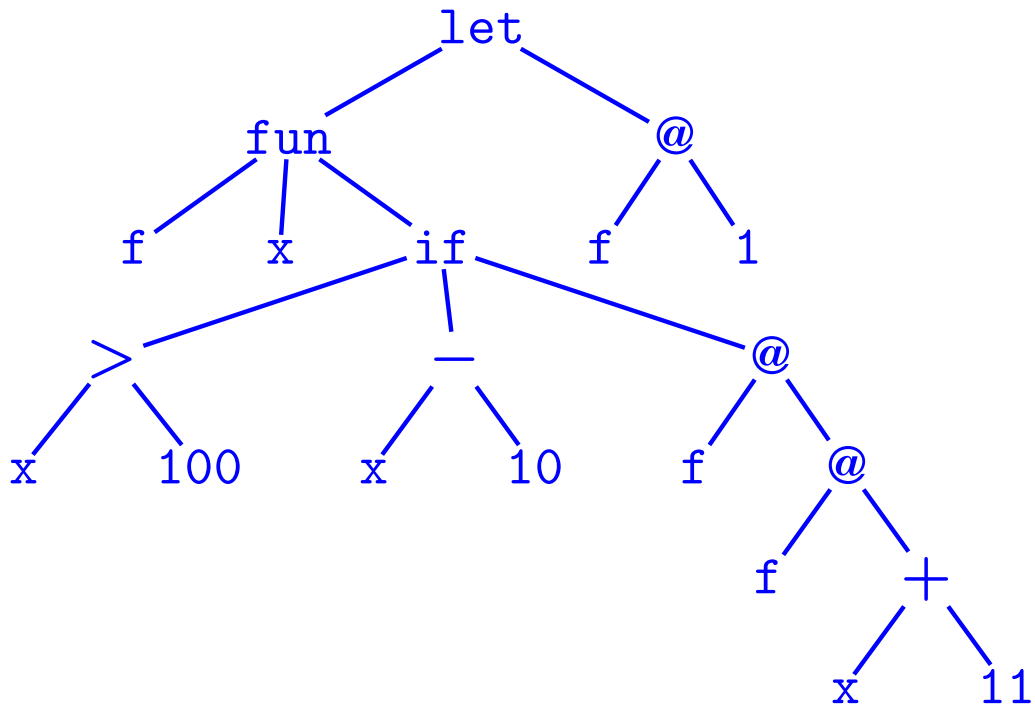
For example, an ML expression:

```
let fun f x =
  if x > 100 then x - 10
  else f ( f ( x + 11 ) )
in f 1 end
(* v a l u e   i s   9 9 *)
```

Abstract syntax: finite rooted trees

- vertexes with n children are labelled by operators expecting n arguments (n-**ary** operators) – in particular leaves are labelled with $\mathbf{0}$-ary (nullary) operators (constants, variables, *etc*)
- label of the root gives the 'outermost form' of the whole phrase

E.g. for the ML expression on Slide 25:



Regular expressions (concrete syntax)

over a given alphabet Σ.

Let Σ' be the 6-element set $\{\epsilon, \emptyset, |, ^*, (,)\}$ (assumed disjoint from Σ)

$$U = (\Sigma \cup \Sigma')^*$$

axioms: $\dfrac{}{a}$ $\qquad$ $\dfrac{}{\epsilon}$ $\qquad$ $\dfrac{}{\emptyset}$

rules: $\dfrac{r}{(r)}$ $\qquad$ $\dfrac{r \quad s}{r|s}$ $\qquad$ $\dfrac{r \quad s}{rs}$ $\qquad$ $\dfrac{r}{r^*}$

(where $a \in \Sigma$ and $r, s \in U$)

Some derivations of regular expressions (assuming $a, b \in \Sigma$)

$$\dfrac{\epsilon \qquad \dfrac{a \qquad \dfrac{b}{b^*}}{ab^*}}{\epsilon|ab^*}$$

$$\dfrac{\dfrac{\epsilon \qquad a}{\epsilon|a} \qquad \dfrac{b}{b^*}}{\epsilon|ab^*}$$

$$\dfrac{\epsilon \qquad \dfrac{\dfrac{a \qquad b}{ab}}{ab^*}}{\epsilon|ab^*}$$

$$\dfrac{\epsilon \qquad \dfrac{a \qquad \dfrac{\dfrac{b}{b^*}}{(b^*)}}{a(b^*)}}{\epsilon|(a(b^*))}$$

$$\dfrac{\dfrac{\epsilon \qquad a}{\epsilon|a} \qquad \dfrac{b}{b^*}}{(\epsilon|a)(b^*)}$$

$$\dfrac{\epsilon \qquad \dfrac{\dfrac{\dfrac{a \qquad b}{ab}}{(ab)}}{(ab)^*}}{\epsilon|((ab)^*)}$$

Regular expressions (abstract syntax)

The 'signature' for regular expression abstract syntax trees (over an alphabet Σ) consists of

- binary operators Union and Concat
- unary operator Star
- nullary operators (constants) Null, Empty and Sym_a (one for each $a \in \Sigma$).

Regular expressions (abstract syntax)

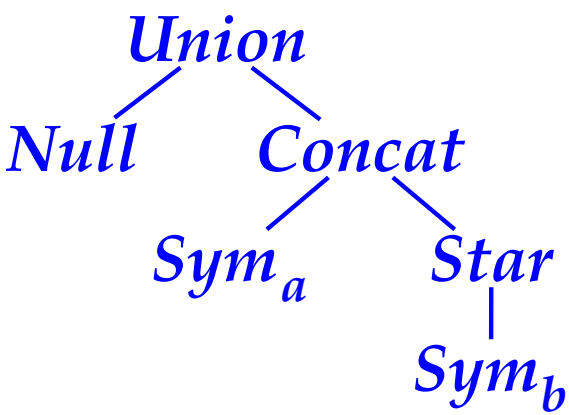
The 'signature' for regular expression abstract syntax trees (over an alphabet Σ) as an ML datatype declaration:

```
datatype 'a RE = Union of ('a RE) * ('a RE)
               | Concat of ('a RE) * ('a RE)
               | Star of 'a RE
               | Null
               | Empty
               | Sym of 'a
```

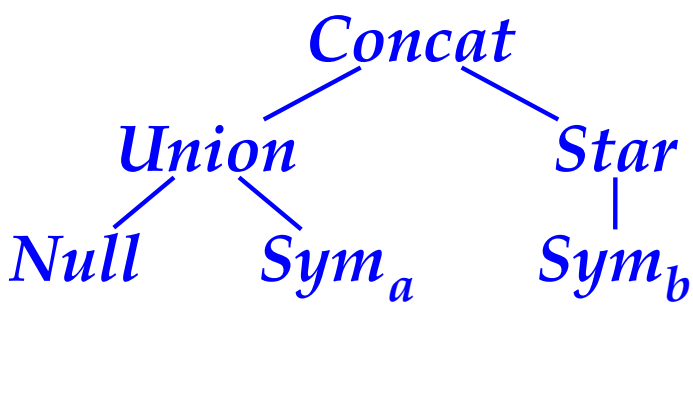
(the type `'a RE` is parameterised by a type variable `'a` standing for the alphabet Σ)

Some abstract syntax trees of regular expressions (assuming $a, b \in \Sigma$)

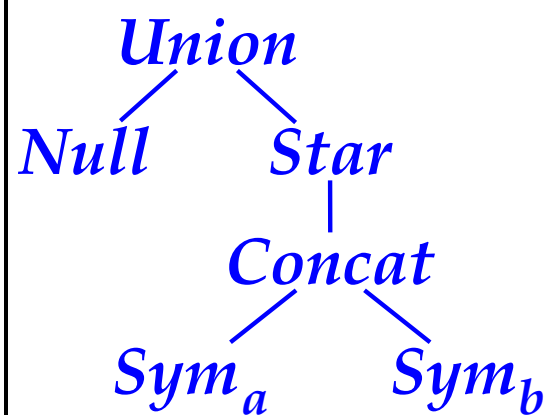
1.



2.



3.



(*cf.* examples on Slide 28)

We will use a textual representation of trees, for example:

1. $Union(Null, Concat(Sym_a, Star(Sym_b)))$
2. $Concat(Union(Null, Sym_a), Star(Sym_b))$
3. $Union(Null, Star(Concat(Sym_a, Sym_b)))$

Relating concrete and abstract syntax

for regular expressions over an alphabet Σ, via an inductively defined relation $\sim$ between strings and trees:

$$\frac{}{a \sim Sym_a} \qquad \frac{}{\epsilon \sim Null} \qquad \frac{}{\emptyset \sim Empty}$$

$$\frac{r \sim R}{(r) \sim R} \qquad \frac{r \sim R \qquad s \sim S}{r|s \sim Union(R,S)}$$

$$\frac{r \sim R \qquad s \sim S}{rs \sim Concat(R,S)} \qquad \frac{r \sim R}{r^* \sim Star(R)}$$

For example:

$$\epsilon|(a(b^*)) \sim Union(Null, Concat(Sym_a, Star(Sym_b)))$$
$$\epsilon|ab^* \sim Union(Null, Concat(Sym_a, Star(Sym_b)))$$
$$\epsilon|ab^* \sim Concat(Union(Null, Sym_a), Star(Sym_b))$$

Thus $\sim$ is a 'many-many' relation between strings and trees.

- **Parsing:** algorithms for producing abstract syntax trees $parse(r)$ from concrete syntax r, satisfying $r \sim parse(r)$.
- **Pretty printing:** algorithms for producing concrete syntax $pp(R)$ from abstract syntax trees R, satisfying $pp(R) \sim R$.

(See CST IB Compiler construction course.)

We can introduce **operator precedence** and **associativity** conventions for concrete syntax and cut down the pairs in the $\sim$ relation to make it single-valued (that is, $r \sim R \;\&\; r \sim R' \Rightarrow R = R'$). For example, for regular expressions we decree:

_* binds more tightly than __, binds more tightly than _|_

So, for example, the only parse of $\epsilon|ab^*$ is the tree

$$Union(Null, Concat(Sym_a, Star(Sym_b)))$$

We also decree that the binary operations of concatenation __ and union _|_ are left associative, so that for example abc parses as

$$Concat(Concat(Sym_a, Sym_b), Sym_c)$$

(However, the union and concatenation operators for regular expressions will always be given a semantics that is associative, so the left-associativity convention is less important than the operator-precedence convention.)

Note: for the rest of the course we adopt these operator-precedence and associativity conventions for regular expressions and refer to abstract syntax trees using concrete syntax.

Matching

Each regular expression r over an alphabet Σ determines a language $L(r) \subseteq \Sigma^*$. The strings u in $L(r)$ are by definition the ones that **match** r, where

- u matches the regular expression a (where $a \in \Sigma$) iff $u = a$
- u matches the regular expression ϵ iff u is the null string ε
- no string matches the regular expression $\emptyset$
- u matches $r|s$ iff it either matches r, or it matches 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.

Inductive definition of matching

$U = \Sigma^* \times \{\text{regular expressions over } \Sigma\}$

abstract syntax trees

axioms: $\dfrac{}{(a,a)}$ $\dfrac{}{(\varepsilon,\epsilon)}$ $\dfrac{}{(\varepsilon,r^*)}$

rules:

$$\frac{(u,r)}{(u,r|s)} \qquad \frac{(u,s)}{(u,r|s)}$$

$$\frac{(v,r) \quad (w,s)}{(vw,rs)} \qquad \frac{(u,r) \quad (v,r^*)}{(uv,r^*)}$$

(No axiom/rule involves the empty regular expression $\emptyset$ – why?)

Examples of matching

Assuming $\Sigma = \{a, b\}$, then:

- $a|b$ is matched by each symbol in Σ
- $b(a|b)^*$ is matched by any string in Σ^* that starts with a 'b'
- $((a|b)(a|b))^*$ is matched by any string of even length in Σ^*
- $(a|b)^*(a|b)^*$ is matched by any string in Σ^*
- $(\epsilon|a)(\epsilon|b)|bb$ is matched by just the strings ε, a, b, ab, and bb
- $\emptyset b|a$ is just matched by a

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, 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, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

(a) The answer to question (a) on Slide 38 is 'yes': algorithms for deciding such pattern-matching questions make use of finite automata. We will see this next.

(b) 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 38 is also 'yes'—the regular expressions defined on Slide 27 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 27. 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 `[a-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 27. The principal example is **complementation**, $\sim(r)$:

u matches $\sim(r)$ iff u *does not* 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 27. 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.

(c) The answer to question (c) on Slide 38 is 'yes' and once again this will be a corollary of the work we do on finite automata.

(d) 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 abstract syntax trees for 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, we will use the Pumping Lemma to see that $\{a^n b^n \mid n \geq 0\}$ is not of this form.

Finite Automata

We will be making use of mathematical models of physical systems called *finite state machines* – of which there are many different varieties. Here we use one particular sort, **finite automata** (singular: finite automaton), to recognise whether or not a string is in a particular language. The key features of this abstract notion of machine are as follows and are illustrated by the example on Slide 44.

- 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 (an element of Σ). In the example $\Sigma = \{a, b\}$ and the only possible transitions from state q_1 are

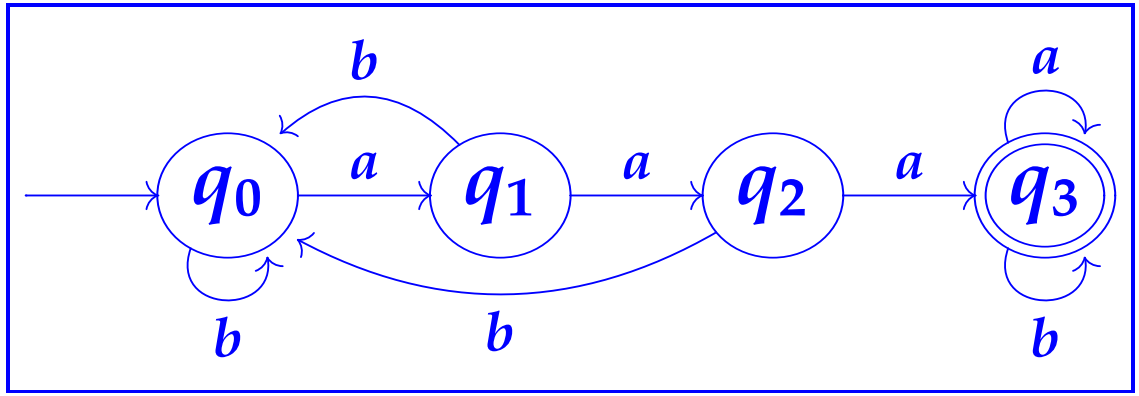
  $$q_1 \xrightarrow{b} q_0 \quad \text{and} \quad q_1 \xrightarrow{a} q_2.$$

  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** (also known as the **initial state**). 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** (also know as **final states**) 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. On the other hand, if there is no path in the graph of transitions from the start state to some accepting state with string of labels equal to u, then u is not in the language accepted by the automaton. This is summed up on Slide 45.

Example of a finite automaton

$M \triangleq$ 

- set of states: $\{q_0, q_1, q_2, q_3\}$
- input alphabet: $\{a, b\}$
- transitions, labelled by input symbols: as indicated by the above directed graph
- start state: q_0
- accepting state(s): q_3

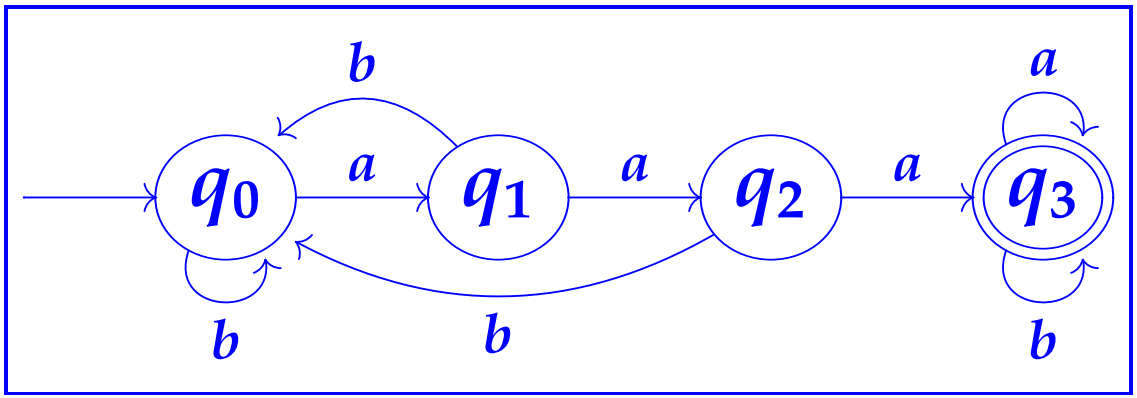
Language accepted by a finite automaton M

- Look at paths in the transition graph from the start state to *some* accepting state.
- Each such path gives a string of input symbols, namely the string of labels on each transition in the path.
- The set of all such strings is by definition **the language accepted by** M, written $L(M)$.

Notation: write $q \xrightarrow{u}{}^{*} q'$ to mean that in the automaton there is a path from state q to state q' whose labels form the string u.

(**N.B.** $q \xrightarrow{\varepsilon}{}^{*} q'$ means $q = q'$.)

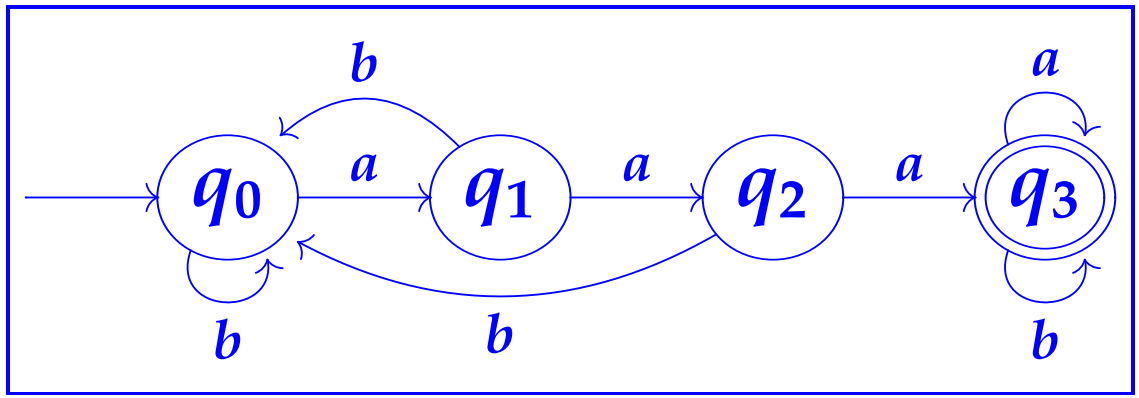
Example of an accepted language

$M \triangleq$ 

For example

- $aaab \in L(M)$, because $q_0 \xrightarrow{aaab}{}^{*} q_3$
- $abaa \notin L(M)$, because $\forall q(q_0 \xrightarrow{abaa}{}^{*} q \Leftrightarrow q = q_2)$

Example of an accepted language

$M \triangleq$ 

Claim:

$$L(M) = L((a|b)^*aaa(a|b)^*)$$

set of all strings matching the regular expression $(a|b)^*aaa(a|b)^*$

(q_i (for $i = 0, 1, 2$) represents the state in the process of reading a string in which the last i symbols read were all as)

Determinism and non-determinism

Slide 49 gives a formal definition of the notion of finite automaton. The reason for the qualification 'non-deterministic' is because in general, for each state $q \in Q$ and each input symbol $a \in \Sigma$, there may be no, one, or many states that can be reached in a single transition labelled a from q; see the example on Slide 50.

We single out as particularly important the case when there is always exactly one next state for a given input symbol in any given state and call such automata **deterministic**: see Slide 51. The finite automaton pictured on Slide 52 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 – see Slide 53. 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).

Non-deterministic finite automaton (NFA)

is by definition a 5-tuple $M = (Q, \Sigma, \Delta, s, F)$, where:

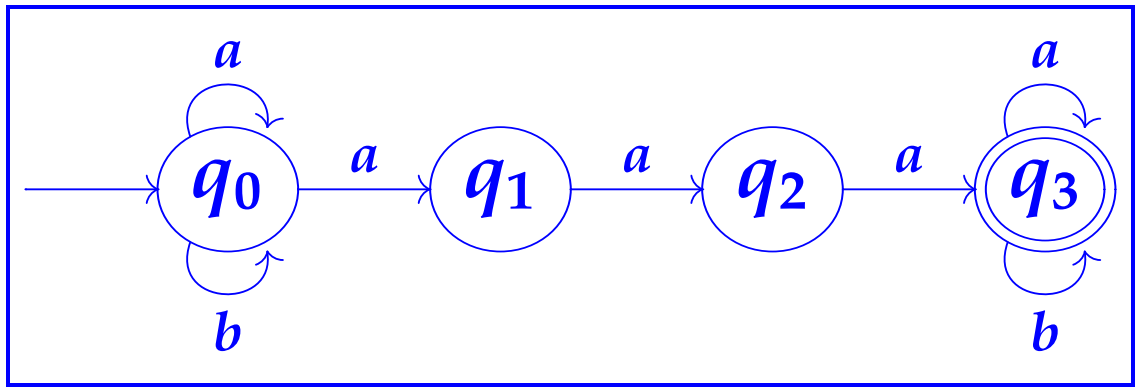
- Q is a finite set (of **states**)
- Σ is a finite set (the alphabet of **input symbols**)
- Δ is a subset of $Q \times \Sigma \times Q$ (the **transition relation**)
- s is an element of Q (the **start state**)
- F is a subset of Q (the **accepting states**)

Notation: write "$q \xrightarrow{a} q'$ in M" to mean $(q, a, q') \in \Delta$.

Example of an NFA

Input alphabet: $\{a, b\}$.

States, transitions, start state, and accepting states as shown:



For example $\{q \mid q_1 \xrightarrow{a} q\} = \{q_2\}$

$\{q \mid q_1 \xrightarrow{b} q\} = \emptyset$

$\{q \mid q_0 \xrightarrow{a} q\} = \{q_0, q_1\}$.

The language accepted by this automaton is the same as for the automaton on Slide 44, namely $\{u \in \{a, b\}^* \mid u$ contains three consecutive a's$\}$.

Deterministic finite automaton (DFA)

A **deterministic finite automaton** (DFA) is an NFA $M = (Q, \Sigma, \Delta, s, F)$ with the property that for each state $q \in Q$ and each input symbol $a \in \Sigma_M$, there is a unique state $q' \in Q$ satisfying $q \xrightarrow{a} q'$.

In a DFA $\Delta \subseteq Q \times \Sigma \times Q$ is the graph of a function $Q \times \Sigma \to Q$, which we write as δ and call the **next-state function**.

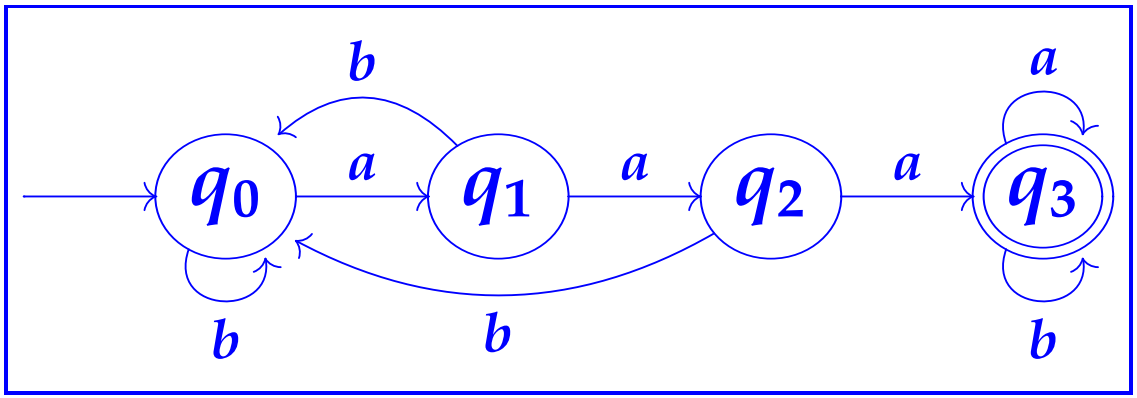
Thus for each (state, input symbol)-pair (q, a), $\delta(q, a)$ is the unique state that can be reached from q by a transition labelled a:

$$\forall q' (q \xrightarrow{a} q' \;\Leftrightarrow\; q' = \delta(q, a))$$

Example of a DFA

with input alphabet $\{a, b\}$

$M \triangleq$

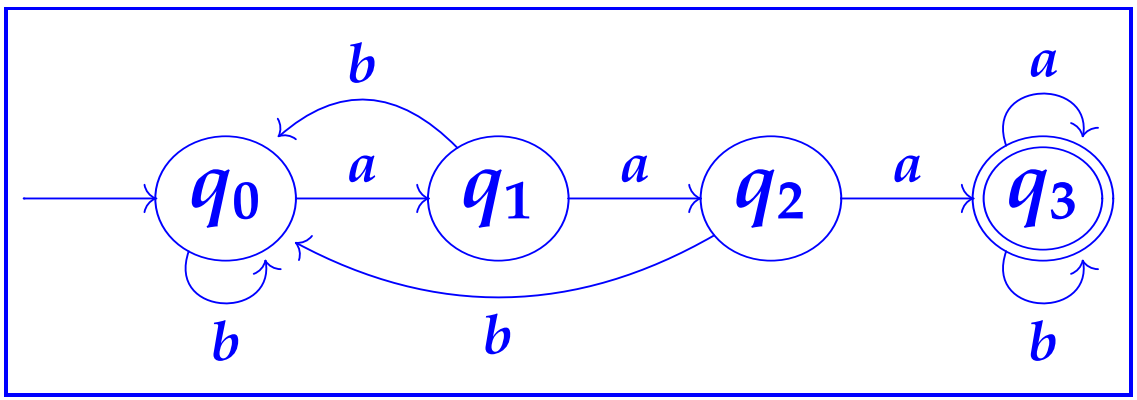


next-state function:

δ	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_3	q_0
q_3	q_3	q_3

Example of an NFA

with input alphabet $\{a, b, c\}$

$M \triangleq$ 

M is non-deterministic, because for example $\{q \mid q_0 \xrightarrow{c} q\} = \emptyset$.

ε-Transitions

When constructing machines for matching strings with regular expressions (as we will do later), 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 55.

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 $q \overset{u}{\Rightarrow} q'$ to indicate that such a sequence exists from state q to state q' in the NFA$^\varepsilon$. Equivalently, $\{(q,u,q') \mid q \overset{u}{\Rightarrow} q'\}$ is the subset of $Q \times \Sigma^* \times Q$ inductively defined by

axioms: $\dfrac{}{(q,\varepsilon,q)}$ and rules: $\dfrac{(q,u,q')}{(q,u,q'')}$ if $q' \xrightarrow{\varepsilon} q''$, $\dfrac{(q,u,q')}{(q,ua,q'')}$ if $q' \xrightarrow{a} q''$ (see Exercise 7)

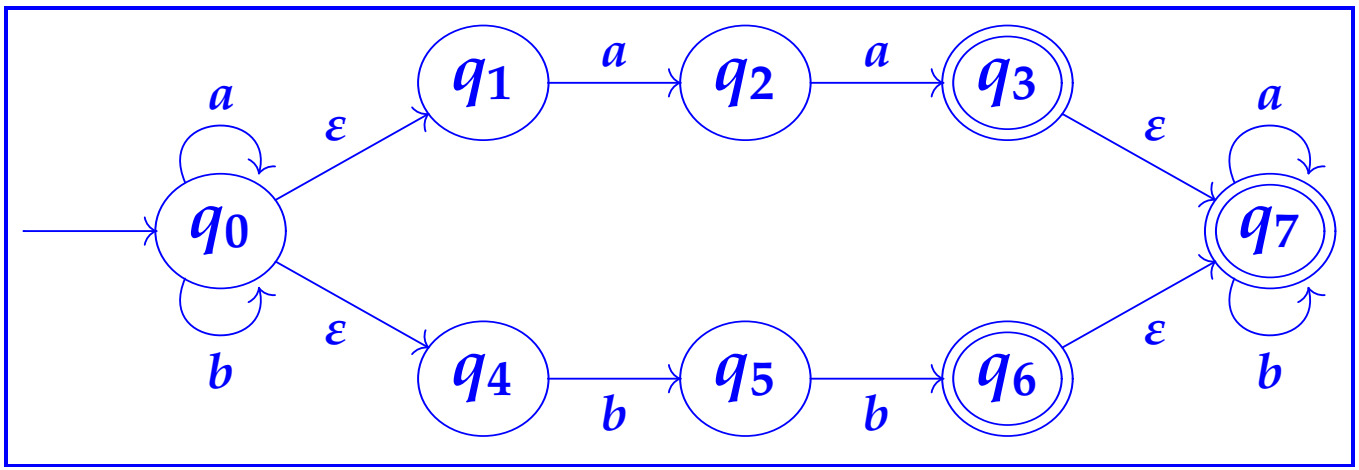
Slide 56 uses the relation $q \overset{u}{\Rightarrow} q'$ to define the language accepted by an NFA$^\varepsilon$. For example, for the NFA$^\varepsilon$ on Slide 55 it is not too hard to see that the language accepted consists of all strings which either contain two consecutive a's or contain two consecutive b's, i.e. the language determined by the regular expression $(a|b)^*(aa|bb)(a|b)^*$.

An **NFA with ε-transitions** (NFA$^\varepsilon$)

$$M = (Q, \Sigma, \Delta, s, F, T)$$

is an NFA $(Q, \Sigma, \Delta, s, F)$ together with a subset $T \subseteq Q \times Q$, called the **ε-transition relation**.

Example:



Notation: write "$q \xrightarrow{\varepsilon} q'$ in M" to mean $(q, q') \in T$.

(**N.B.** for NFA$^\varepsilon$s, we always assume $\varepsilon \notin \Sigma$.)

Language accepted by an NFA$^{\varepsilon}$

$M = (Q, \Sigma, \Delta, s, F, T)$

- Look at paths in the transition graph (including ε-transitions) from start state to *some* accepting state.
- Each such path gives a string in Σ^*, namely the string of non-ε labels that occur along the path.
- The set of all such strings is by definition **the language accepted by** M, written $L(M)$.

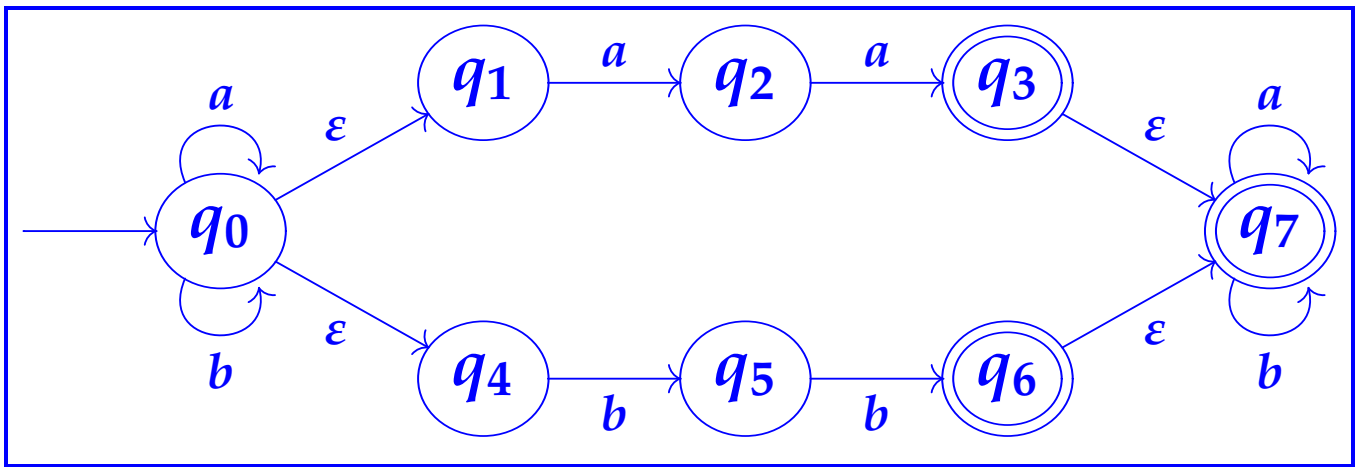
Notation: write $q \stackrel{u}{\Rightarrow} q'$ to mean that there is a path in M from state q to state q' whose non-ε labels form the string $u \in \Sigma^*$.

An **NFA with ε-transitions** (NFA$^\varepsilon$)

$$M = (Q, \Sigma, \Delta, s, F, T)$$

is an NFA $(Q, \Sigma, \Delta, s, F)$ together with a subset $T \subseteq Q \times Q$, called the **ε-transition relation**.

Example:



For this NFA$^\varepsilon$ we have, e.g.: $q_0 \stackrel{aa}{\Rightarrow} q_2$, $q_0 \stackrel{aa}{\Rightarrow} q_3$ and $q_0 \stackrel{aa}{\Rightarrow} q_7$.

In fact the language of accepted strings is equal to the set of strings matching the regular expression $(a|b)^*(aa|bb)(a|b)^*$.

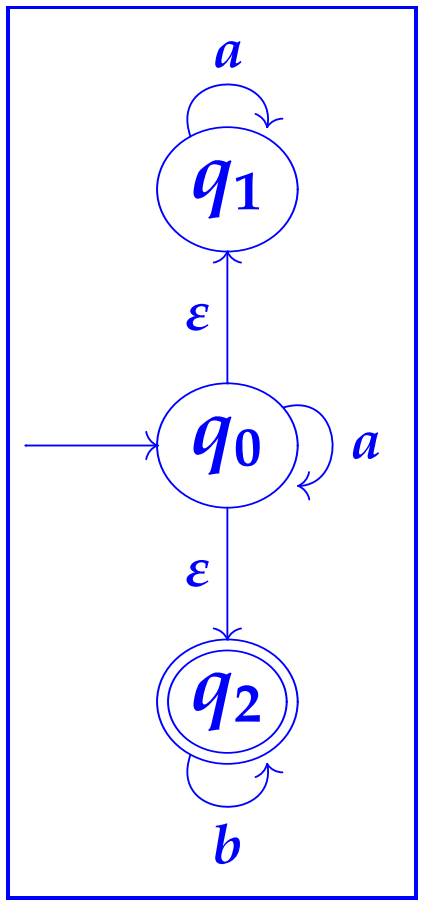
The subset construction for NFA$^\varepsilon$s

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 sets of strings accepted 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 59 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 of states of M. Given two such subsets, S and S' say, there is a transition $S \xrightarrow{a} S'$ in PM just in case S' consists of all the M-states q' reachable from states q in S via the $\cdot \stackrel{a}{\Rightarrow} \cdot$ relation defined on Slide 56, 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



next-state function for PM

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_1\}$	$\{q_1\}$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$
$\{q_1, q_2\}$	$\{q_1\}$	$\{q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_2\}$

Theorem. For each NFA$^\varepsilon$ $M = (Q, \Sigma, \Delta, s, F, T)$ there is a DFA $PM = (\mathcal{P}(Q), \Sigma, \delta, s', F')$ accepting exactly the same strings as M, i.e. with $L(PM) = L(M)$.

Definition of PM:

- set of states is the powerset $\mathcal{P}(Q) = \{S \mid S \subseteq Q\}$ of the set Q of states of M
- same input alphabet Σ as for M
- next-state function maps each $(S, a) \in \mathcal{P}(Q) \times \Sigma$ to $\delta(S, a) \triangleq \{q' \in Q \mid \exists q \in S.\ q \stackrel{a}{\Rightarrow} q' \text{ in } M\}$
- start state is $s' \triangleq \{q' \in Q \mid s \stackrel{\varepsilon}{\Rightarrow} q'\}$
- subset of accepting states is $F' \triangleq \{S \in \mathcal{P}(Q) \mid S \cap F \neq \emptyset\}$

To prove the theorem we show that $L(M) \subseteq L(PM)$ and $L(PM) \subseteq L(M)$.

Proof that $L(M) \subseteq L(PM)$

Consider the case of ε first: if $\varepsilon \in L(M)$, then $s \stackrel{\varepsilon}{\Rightarrow} q$ for some $q \in F$, hence $s' \in F'$ and thus $\varepsilon \in L(PM)$.

Now given any non-null string $u = a_1 a_2 \ldots a_n$, if $u \in L(M)$, then there is a sequence of transitions in M of the form

$$s \stackrel{a_1}{\Rightarrow} q_1 \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} q_n \in F \tag{1}$$

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

$$s' \xrightarrow{a_1} S_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} S_n \tag{2}$$

where $S_1 = \delta(s', a_1)$, $S_2 = \delta(S_1, a_2)$, etc. By definition of δ (Slide 60), from (1) we deduce

$$q_1 \in \delta(s', a_1) = S_1, \text{ hence } q_2 \in \delta(S_1, a_2) = S_2, \ldots, \text{ hence } q_n \in \delta(S_{n-1}, a_n) = S_n.$$

Therefore $S_n \in F'$ (because $q_n \in S_n \cap F$). So (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 $s' \in F'$ and so there is some $q \in s'$ with $q \in F$, i.e. $s \stackrel{\varepsilon}{\Rightarrow} q \in F$ and thus $\varepsilon \in L(M)$.

Now given any non-null string $u = a_1 a_2 \ldots a_n$, if $u \in L(PM)$, then there is a sequence of transitions in PM of the form (2) with $S_n \in F'$, i.e. with S_n containing some $q_n \in F$. Now since $q_n \in S_n = \delta(S_{n-1}, a_n)$, by definition of δ there is some $q_{n-1} \in S_{n-1}$ with $q_{n-1} \stackrel{a_n}{\Rightarrow} q_n$ in M. Then since $q_{n-1} \in S_{n-1} = \delta(S_{n-2}, a_{n-1})$, there is some $q_{n-2} \in S_{n-2}$ with $q_{n-2} \stackrel{a_{n-1}}{\Rightarrow} q_{n-1}$. Working backwards in this way we can build up a sequence of transitions like (1) until, at the last step, from the fact that $q_1 \in S_1 = \delta(s', a_1)$ we deduce that $s \stackrel{a_1}{\Rightarrow} q_1$. So we get a sequence of transitions (1) with $q_n \in F$, and hence u is accepted by M. □

Regular Languages

Kleene's Theorem

Definition. A language is **regular** iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton M.

Theorem.

(a) For any regular expression r, the set $L(r)$ of strings matching r is a regular language.

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

Kleene part (a): from regular expressions to automata

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 60 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 *induction on the size (= number of vertices) of regular expression abstract syntax trees*, as indicated on the next slide. 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 abstract syntax trees 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.

(i) **Base cases:** show that $\{a\}$, $\{\varepsilon\}$ and $\emptyset$ are regular languages.

(ii) **Induction step for $r_1|r_2$:** given NFA$^\varepsilon$s M_1 and M_2, construct an NFA$^\varepsilon$ $Union(M_1, M_2)$ satisfying

$$L(Union(M_1, M_2)) = \{u \mid u \in L(M_1) \vee u \in L(M_2)\}$$

Thus if $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$, then $L(r_1|r_2) = L(Union(M_1, M_2))$.

(iii) **Induction step for $r_1 r_2$:** given NFA$^\varepsilon$s M_1 and M_2, construct an NFA$^\varepsilon$ $Concat(M_1, M_2)$ satisfying

$$L(Concat(M_1, M_2)) = \{u_1 u_2 \mid u_1 \in L(M_1) \;\&\; u_2 \in L(M_2)\}$$

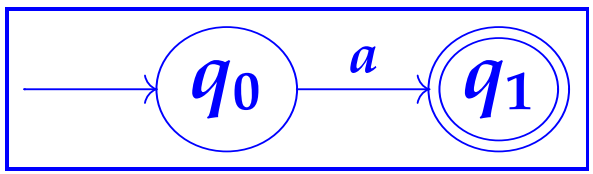
Thus $L(r_1 r_2) = L(Concat(M_1, M_2))$ when $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$.

(iv) **Induction step for r^*:** given NFA$^\varepsilon$ M, construct an NFA$^\varepsilon$ $Star(M)$ satisfying

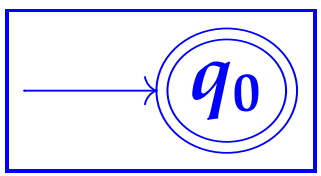
$$L(Star(M)) = \{u_1 u_2 \ldots u_n \mid n \geq 0 \text{ and each } u_i \in L(M)\}$$

Thus $L(r^*) = L(Star(M))$ when $L(r) = L(M)$.

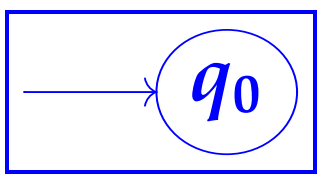
NFAs for regular expressions a, ϵ, $\emptyset$



just accepts the one-symbol string a

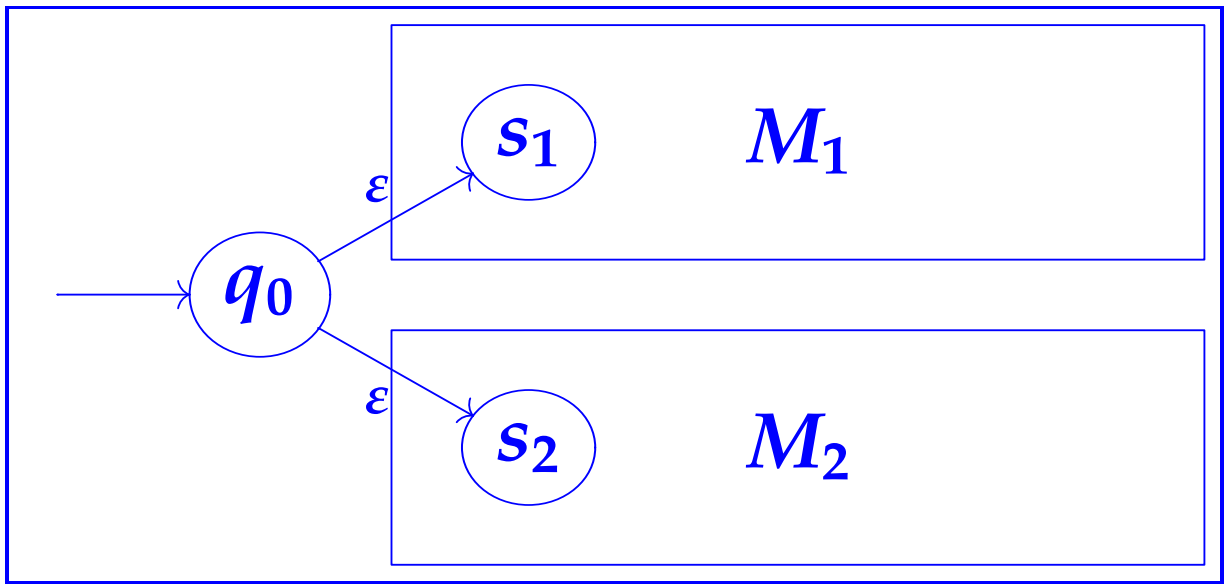


just accepts the null string, ε



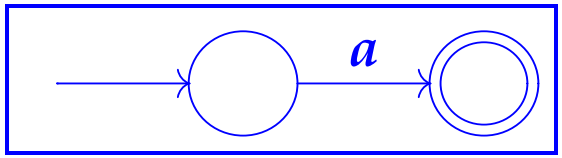
accepts no strings

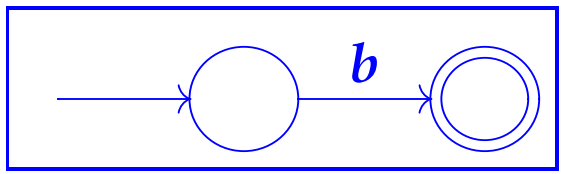
$Union(M_1, M_2)$

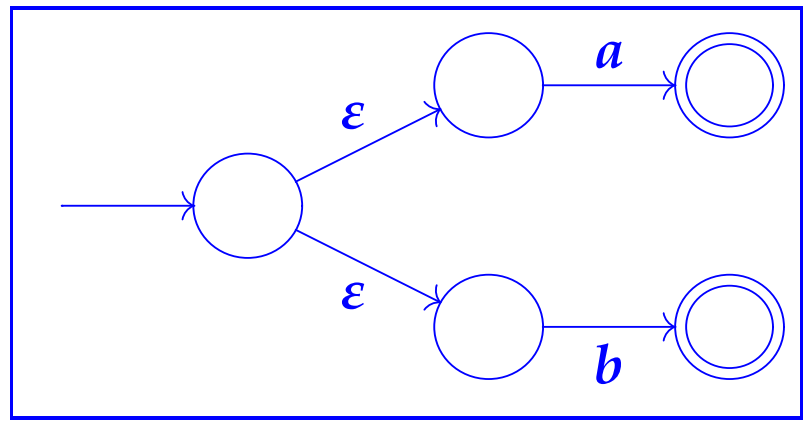


accepting states = union of accepting states of M_1 and M_2

For example,

if $M_a =$ 

and $M_b =$ 

then $Union(M_a, M_b) =$ 

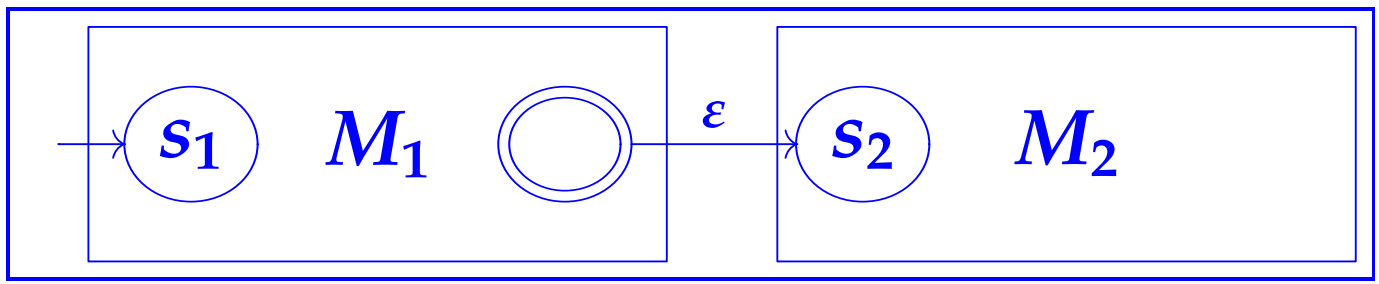
Induction step for $r_1|r_2$

Given NFA$^\varepsilon$s $M_1 = (Q_1, \Sigma, \Delta_1, s_1, T_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, s_2, T_2)$, the construction of $\mathit{Union}(M_1, M_2)$ is pictured on Slide 68. First, renaming states if necessary, we assume that $Q_1 \cap Q_2 = \emptyset$. Then the states of $\mathit{Union}(M_1, M_2)$ are all the states in either Q_1 or Q_2, together with a new state, called q_0 say. The start state of $\mathit{Union}(M_1, M_2)$ is this q_0 and its set of accepting states is the union $F_1 \cup F_2$ of the sets of accepting states in M_1 and M_2. Finally, the transitions of $\mathit{Union}(M_1, M_2)$ are given by all those in either M_1 or M_2, together with two new ε-transitions out of q_0, one to the start states s_1 of M_1 and one to the start state s_2 of M_2.

Thus if $u \in L(M_1)$, i.e. if we have $s_1 \stackrel{u}{\Rightarrow} q_1$ for some $q_1 \in F_1$, then we get $q_0 \stackrel{\varepsilon}{\longrightarrow} s_1 \stackrel{u}{\Rightarrow} q_1$ showing that $u \in L(\mathit{Union}(M_1, M_2))$. Similarly for M_2. So $L(\mathit{Union}(M_1, M_2))$ contains the union of $L(M_1)$ and $L(M_2)$. Conversely if u is accepted by $\mathit{Union}(M_1, M_2)$, there is a transition sequence $q_0 \stackrel{u}{\Rightarrow} q$ with $q \in F_1$ or $q \in F_2$. 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 (*because we assumed that* Q_1 *and* Q_2 *are disjoint*) finishing in an acceptable state for that one. So if $u \in L(\mathit{Union}(M_1, M_2))$, then either $u \in L(M_1)$ or $u \in L(M_2)$. So we do indeed have

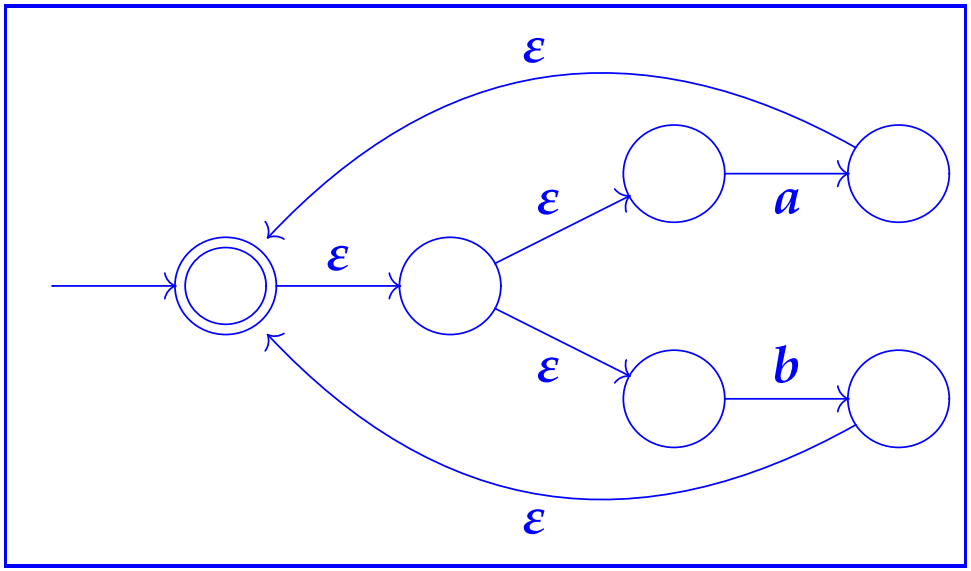
$$L(\mathit{Union}(M_1, M_2)) = \{u \mid u \in L(M_1) \vee u \in L(M_2)\}$$

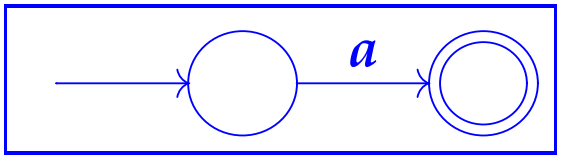
$Concat(M_1, M_2)$

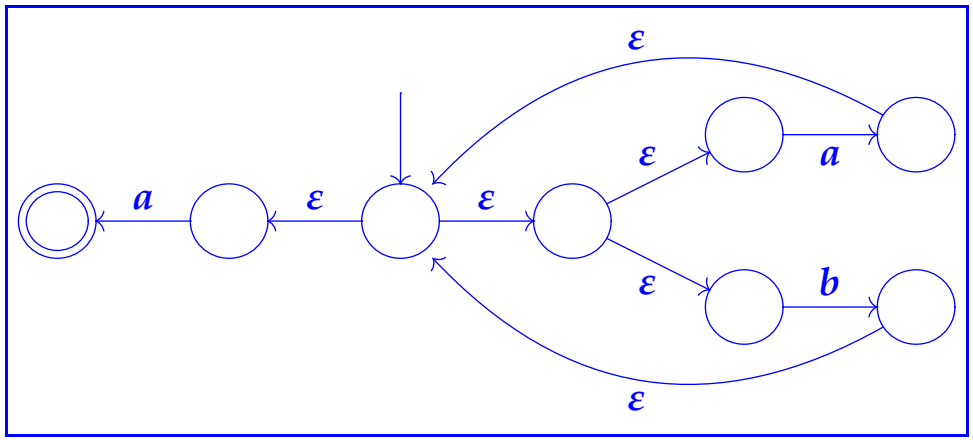


accepting states are those of M_2

For example,

if $M_1 =$ 

and $M_2 =$ 

then $Concat(M_1, M_2) =$ 

Induction step for $r_1 r_2$

Given NFA$^\varepsilon$s $M_1 = (Q_1, \Sigma, \Delta_1, s_1, T_1)$ and $M_2 = (Q_2, \Sigma, \Delta_2, s_2, T_2)$, the construction of $Concat(M_1, M_2)$ is pictured on Slide 71. First, renaming states if necessary, we assume that $Q_1 \cap Q_2 = \emptyset$. Then the set of states of $Concat(M_1, M_2)$ is $Q_1 \cup Q_2$. The start state of $Concat(M_1, M_2)$ is the start state s_1 of M_1. The set of accepting states of $Concat(M_1, M_2)$ is the set F_2 of 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 s_2 of M_2 (only one such new transition is shown in the picture).

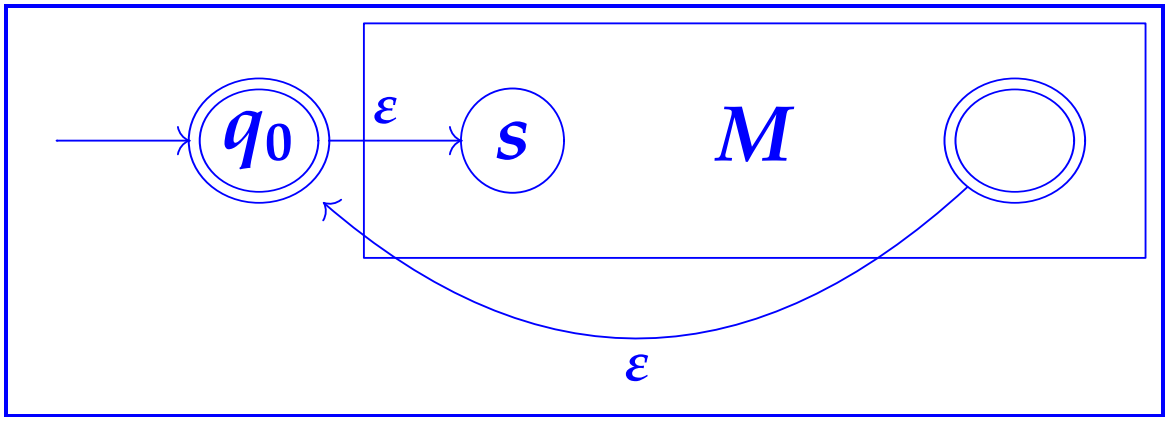
Thus if $u_1 \in L(M_1)$ and $u_2 \in L(M_2)$, there are transition sequences $s_1 \stackrel{u_1}{\Rightarrow} q_1$ in M_1 with $q_1 \in F_1$, and $s_2 \stackrel{u_2}{\Rightarrow} q_2$ in M_2 with $q_2 \in F_2$. These combine to yield

$$s_1 \stackrel{u_1}{\Rightarrow} q_1 \stackrel{\varepsilon}{\longrightarrow} s_2 \stackrel{u_2}{\Rightarrow} q_2$$

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

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

$Star(M)$

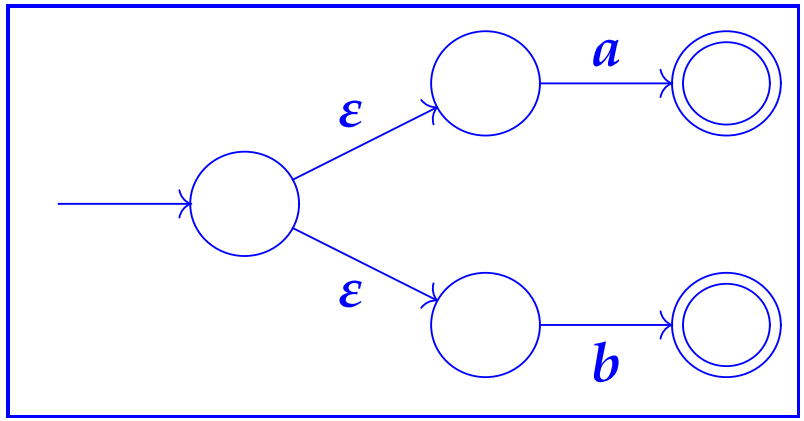


the only accepting state of $Star(M)$ is q_0

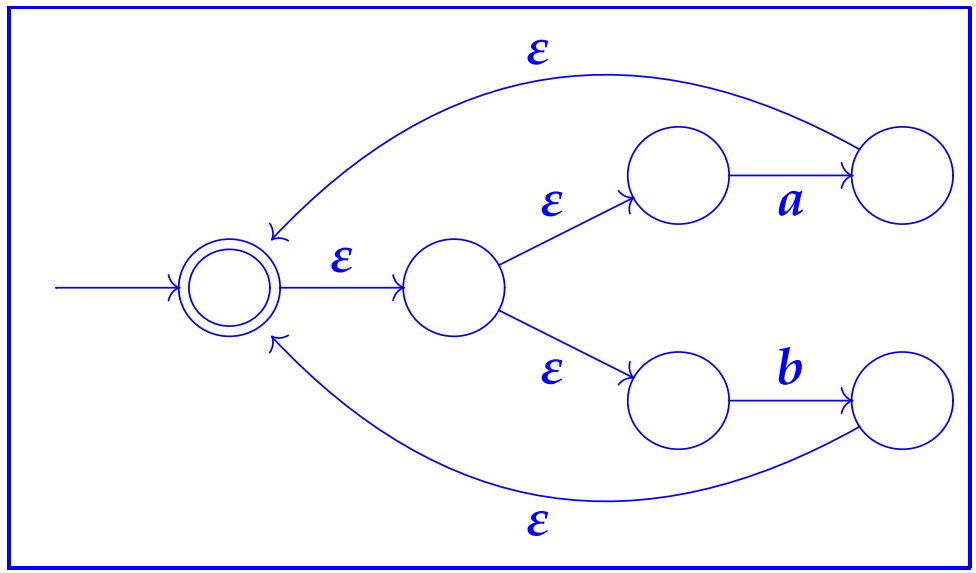
(N.B. doing without q_0 by just looping back to s
and making that accepting won't work – Exercise 4.1.)

For example,

if $M =$



then $Star(M) =$



Induction step for r^*

Given an NFA$^\varepsilon$ $M = (Q, \Sigma, \Delta, s, T)$, the construction of $Star(M)$ is pictured on Slide 74. 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

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

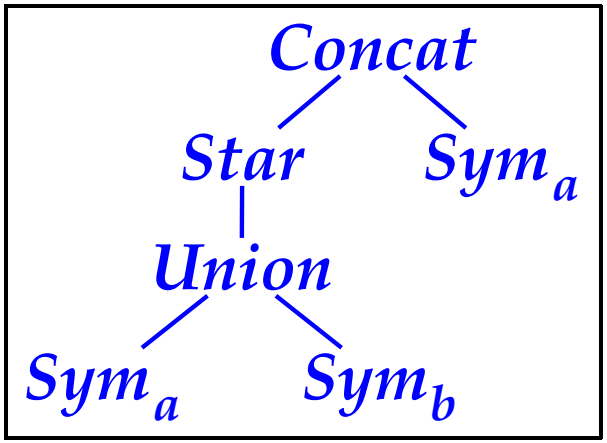
□

This completes the proof of part (a) of Kleene's Theorem (Slide 64). Slide 77 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.

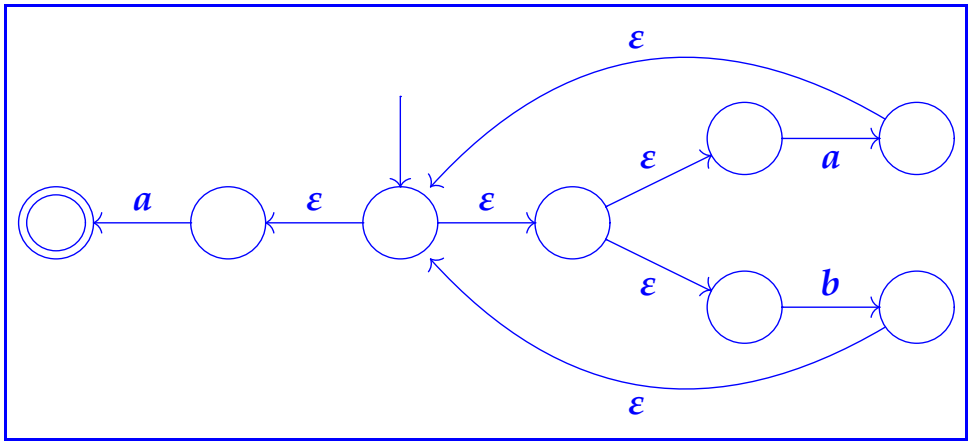
Example

Regular expression $(a|b)^*a$

whose abstract syntax tree is



is mapped to the NFA$^\varepsilon$ $Concat(Star(Union(M_a, M_b)), M_a) =$



(*cf.* Slides 69, 72 and 75).

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, 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, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

Decidability of matching

We now have a positive answer to question (a) on Slide 38. Given string u and regular expression r:

- construct an NFA$^\varepsilon$ M satisfying $L(M) = L(r)$;
- in PM (the DFA obtained by the subset construction, Slide 60) carry out the sequence of transitions corresponding to u from the start state to some state q (because PM is deterministic, there is a unique such transition sequence);
- check whether q is accepting or not: if it is, then $u \in L(PM) = L(M) = L(r)$, so u matches r; otherwise $u \notin L(PM) = L(M) = L(r)$, so u does not match r.

(The subset construction produces an exponential blow-up of the number of states: PM has 2^n states if M has n. This makes the method described above potentially inefficient – more efficient algorithms exist that don't construct the whole of PM.)

Kleene's Theorem

Definition. A language is **regular** iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton M.

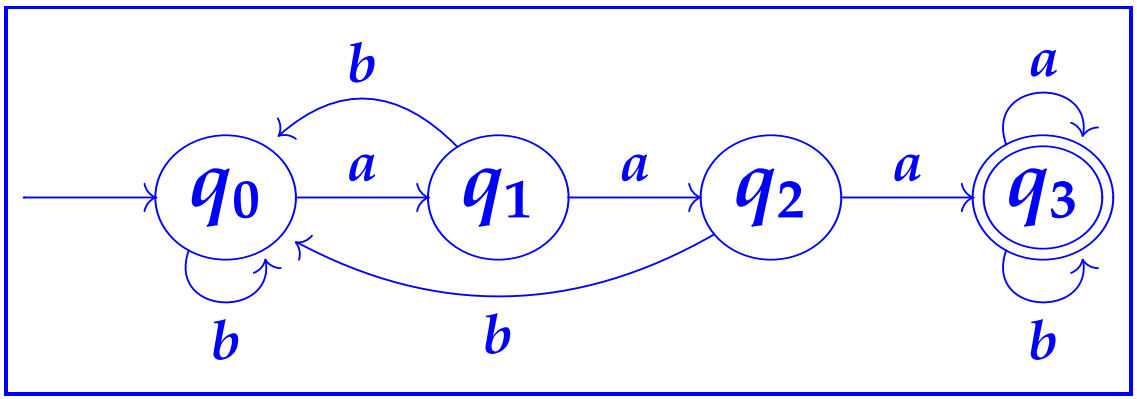
Theorem.

(a) For any regular expression r, the set $L(r)$ of strings matching r is a regular language.

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

Example of a regular language

Recall the example DFA we used earlier:

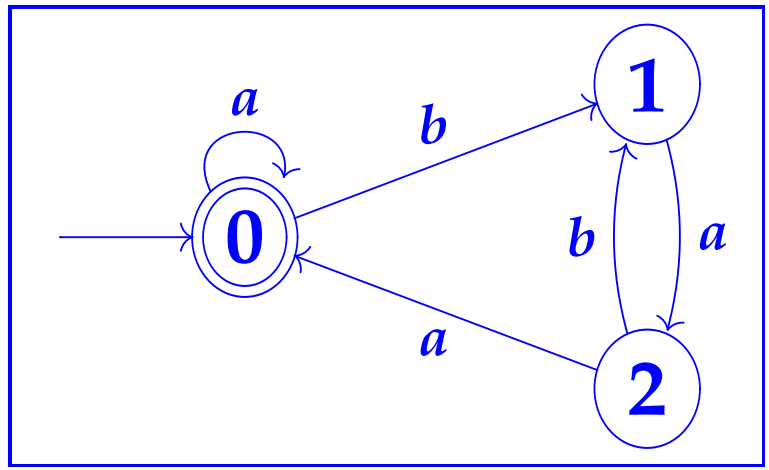
$M \triangleq$ 

In this case it's not hard to see that $L(M) = L(r)$ for

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

Example

$M \triangleq$



$L(M) = L(r)$ for which regular expression r?

Guess: $r = a^*|a^*b(ab)^*aaa^*$

WRONG! since $baabaa \in L(M)$
but $baabaa \notin L(a^*|a^*b(ab)^*aaa^*)$

We need an algorithm for constructing a suitable r for each M (plus a proof that it is correct).

Kleene part (b): from automata to regular expressions

Given any DFA $M = (Q, \Sigma, \delta, s, F)$, we have to find a regular expression r (over the alphabet Σ 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 84. Note that if we can find the regular expressions $r^S_{q,q'}$ mentioned in the lemma (for any choice of $S \subseteq Q$ and $q, q' \in Q$), then the problem is solved. For taking S to be the whole of Q and q to be the start state s, 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 $M = (Q, \Sigma, \Delta, s, F)$, for each subset $S \subseteq Q$ and each pair of states $q, q' \in Q$, there is a regular expression $r^S_{q,q'}$ satisfying

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

Hence if the subset F of accepting states has k distinct elements, $q_1, \ldots, q_k$ say, then $L(M) = L(r)$ with $r \triangleq r_1 | \cdots | r_k$ where

$$r_i = r^Q_{s,q_i} \qquad (i = 1, \ldots, k)$$

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

Proof of the Lemma on Slide 84

The regular expression $r^S_{q,q'}$ can be constructed by induction on the number of elements in the subset S.

Base case, S 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

$$\{u \mid q \xrightarrow{u}{}^{*} q' \text{ with } \textit{no} \text{ intermediate states in the sequence of transitions}\}.$$

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

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

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

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

[**Notation:** given sets X and Y, we write $X \setminus Y$ for the set $\{x \in X \mid x \notin Y\}$ of elements of X that are not in Y and call it the *relative complement* of X by Y.]

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

$$r_1 \triangleq r_{q,q'}^{S\setminus\{q_0\}} \qquad r_2 \triangleq r_{q,q_0}^{S\setminus\{q_0\}} \qquad r_3 \triangleq r_{q_0,q_0}^{S\setminus\{q_0\}}, \qquad r_4 \triangleq r_{q_0,q'}^{S\setminus\{q_0\}}$$

Consider the regular expression

$$r \triangleq r_1 | r_2 (r_3)^* r_4$$

Clearly every string matching r is in the set

$$\{u \mid q \xrightarrow{u}{}^* q' \text{ with all intermediate states in the sequence of transitions in } S\}.$$

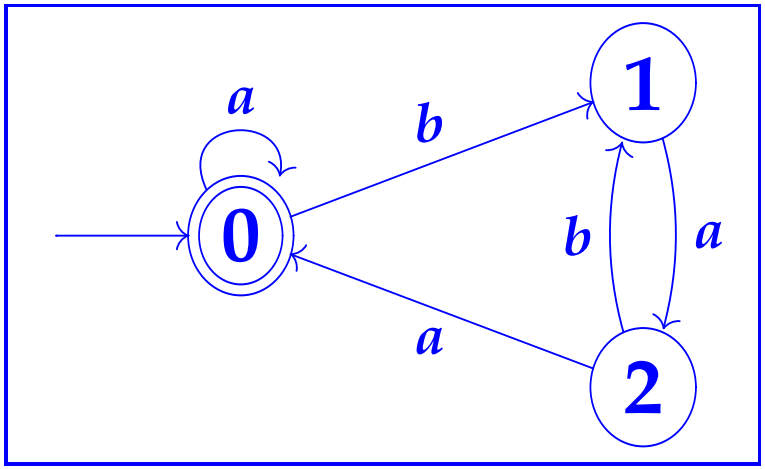
Conversely, if u is in this set, consider the number of times the sequence of transitions $q \xrightarrow{u}{}^* q'$ passes through state q_0. If this number is zero then $u \in L(r_1)$ (by definition of r_1). Otherwise this number is $k \geq 1$ and the sequence splits into $k+1$ pieces: the first piece 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 $r_{q,q'}^S$ to be this regular expression $r = r_1 | r_2 (r_3)^* r_4$. □

An example

We give an example to illustrate the construction of regular expressions from automata that is inherent in the above proof of part (b) of Kleene's Theorem. The example also demonstrates that we do not have to pursue the inductive construction of the regular expression to the bitter end (the base case $S = \emptyset$): often it is possible to find some of the regular expressions $r^S_{q,q'}$ one needs by *ad hoc* arguments – but if in doubt, *use the algorithm*.

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 S. A good rule of thumb is: *choose a state that disconnects the automaton as much as possible.*

$M \triangleq$



By direct inspection we have:

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

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

(we don't need the unfilled entries in the tables)

Consider the NFA shown on Slide 88. Since the start state is $\mathbf{0}$ and this is also the only accepting state, the language of accepted strings is that determined by the regular expression $r_{0,0}^{\{0,1,2\}}$. Choosing to remove state $\mathbf{1}$ from the state set, we have

$$L(r_{0,0}^{\{0,1,2\}}) = L(r_{0,0}^{\{0,2\}} | r_{0,1}^{\{0,2\}} (r_{1,1}^{\{0,2\}})^* r_{1,0}^{\{0,2\}}) \tag{3}$$

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

$$L(r_{1,1}^{\{0,2\}}) = L(r_{1,1}^{\{0\}} | r_{1,2}^{\{0\}} (r_{2,2}^{\{0\}})^* r_{2,1}^{\{0\}})$$
$$L(r_{1,0}^{\{0,2\}}) = L(r_{1,0}^{\{0\}} | r_{1,2}^{\{0\}} (r_{2,2}^{\{0\}})^* r_{2,0}^{\{0\}})$$

These regular expressions can all be determined by inspection, as shown on Slide 88. Thus $L(r_{1,1}^{\{0,2\}}) = L(\varepsilon | a(\varepsilon)^*(a^*b))$ and it's not hard to see that this is equal to $L(\varepsilon | aa^*b)$. Similarly $L(r_{1,0}^{\{0,2\}}) = L(\emptyset | a(\varepsilon)^*(aa^*))$ 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 88. (Clearly, one could simplify this to a smaller, but equivalent regular expression, but we do not bother to do so.)

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, 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, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

Regular languages are closed under complementation

Lemma. 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 92. Then $\{u \in \Sigma^* \mid u \notin L\}$ is the set of strings accepted by $Not(M)$ and hence is regular. □

[**N.B.** If one applies the construction on Slide 92 (interchanging the role of accepting & non-accepting states) to a *non-deterministic* finite automaton N, then in general $L(Not(N))$ is not equal to $\{u \in \Sigma^* \mid u \notin L(N)\}$ – see Exercise 4.5.]

We saw on slide 79 that part (a) of Kleene's Theorem allows us to answer question (a) on Slide 38. Now that we have proved the other half of the theorem, we can say more about question (b) on that slide. In particular, it is a consequence of Kleene's Theorem plus the above lemma that for each regular expression r over an alphabet Σ, there is a regular expression $\sim r$ that determines via matching the complement of the language determined by r:

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

To see this, 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.

$Not(M)$

Given DFA $M = (Q, \Sigma, \delta, s, F)$,
then $Not(M)$ is the DFA with

- set of states $= Q$
- input alphabet $= \Sigma$
- next-state function $= \delta$
- start state $= s$
- accepting states $= \{q \in Q \mid q \notin F\}$.

(i.e. we just reverse the role of accepting/non-accepting and leave everything else the same)

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

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

Regular languages are closed under intersection

Theorem. If L_1 and L_2 are a regular languages over an alphabet Σ, then their intersection $L_1 \cap L_2 = \{u \in \Sigma^* \mid u \in L_1 \,\&\, u \in L_2\}$ is also regular.

Proof. Note that $L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2))$

(*cf.* de Morgan's Law: $p \,\&\, q = \neg(\neg p \vee \neg q)$).

So if $L_1 = L(M_1)$ and $L_2 = L(M_2)$ for DFAs M_1 and M_2, then $L_1 \cap L_2 = L(Not(PM))$ where M is the NFA$^\varepsilon$ $Union(Not(M_1), Not(M_2))$. □

[It is not hard to directly construct a DFA $And(M_1, M_2)$ from M_1 and M_2 such that $L(And(M_1, M_2)) = L(M_1) \cap L(M_2)$ – see Exercise 4.7.]

Regular languages are closed under intersection

Corollary: given regular expressions r_1 and r_2,there is a regular expression, which we write as $r_1 \,\&\, r_2$, such that

> a string u matches $r_1 \,\&\, r_2$ iff it matches both r_1 and r_2.

Proof. By Kleene (a), $L(r_1)$ and $L(r_2)$ are regular languages and hence by the theorem, so is $L(r_1) \cap L(r_2)$. Then we can use Kleene (b) to construct a regular expression $r_1 \,\&\, r_2$ with $L(r_1 \,\&\, r_2) = L(r_1) \cap L(r_2)$. □

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, 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, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

Equivalent regular expressions

Definition. Two regular expressions r and s are said to be **equivalent** if $L(r) = L(s)$, that is, they determine exactly the same sets of strings via matching.

For example, are $b^*a(b^*a)^*$ and $(a|b)^*a$ equivalent?

Answer: yes (Exercise 2.3)

How can we decide all such questions?

Note that $L(r) = L(s)$

iff $L(r) \subseteq L(s)$ and $L(s) \subseteq L(r)$
iff $(\Sigma^* \setminus L(r)) \cap L(s) = \emptyset = (\Sigma^* \setminus L(s)) \cap L(r)$
iff $L((\sim r)\,\&\,s) = \emptyset = L((\sim s)\,\&\,r)$
iff $L(M) = \emptyset = L(N)$

where M and N are DFAs accepting the sets of strings matched by the regular expressions $(\sim r)\,\&\,s$ and $(\sim s)\,\&\,r$ respectively.

So to decide equivalence for regular expressions it suffices to

check, given any given DFA M, whether or not it accepts some string.

Note that the number of transitions needed to reach an accepting state in a finite automaton is bounded by the number of states (we can remove loops from longer paths). So we only have to check finitely many strings to see whether or not $L(M)$ is empty.

The Pumping Lemma

Some questions

(a) Is there an algorithm which, given a string u and a regular expression r, 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, computes whether or not they are **equivalent**, in the sense that $L(r)$ and $L(s)$ are equal sets?

(d) Is every language (subset of Σ^*) of the form $L(r)$ for some r?

In the context of programming languages, a typical example of a regular language (Slide 64) 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 101 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).

The intuitive reason why the languages listed on Slide 101 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 102.

Examples of languages that are not regular

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

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 $|w| \geq \ell$ can be expressed as a concatenation of three strings, $w = u_1 v u_2$, where u_1, v and u_2 satisfy:

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

Proving the Pumping Lemma

Since L is regular, it is equal to the set $L(M)$ of strings accepted by some DFA $M = (Q, \Sigma, \delta, s, F)$. Then *we can take the number ℓ mentioned on Slide 102 to be the number of states in Q.* For suppose $w = a_1 a_2 \ldots a_n$ with $n \geq \ell$. If $w \in L(M)$, then there is a transition sequence as shown at the top of Slide 104. Then w can be split into three pieces as shown on that slide. Note that by choice of i and j, $|v| = j - i \geq 1$ and $|u_1 v| = j \leq \ell$. 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 104, 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. $u_1 v^n u_2 \in L$. □

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

Remark. 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 4.3, 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 102? 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 $|w| \geq \ell$ for which we have to check the condition!

Suppose $L = L(M)$ for a DFA $M = (Q, \Sigma, \delta, s, F)$. Taking ℓ to be the number of elements in Q, if $n \geq \ell$, then in

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

$q_0, \ldots, q_\ell$ 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

$$s = q_0 \xrightarrow{u_1\,*} q_i \overset{v}{=} q_j \xrightarrow{u_2\,*} q_n \in F$$

where

$$u_1 \triangleq a_1 \ldots a_i \qquad v \triangleq a_{i+1} \ldots a_j \qquad u_2 \triangleq a_{j+1} \ldots a_n$$

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

no matter how w is split into three, $w = u_1 v u_2$,
with $|u_1 v| \leq \ell$ and $|v| \geq 1$, there is some $n \geq 0$
for which $u_1 v^n u_2$ is *not* in L

} (†)

Examples

None of the following three languages are regular:

(i) $L_1 \triangleq \{a^n b^n \mid n \geq 0\}$

[For each $\ell \geq 1$, $a^\ell b^\ell \in L_1$ is of length $\geq \ell$ and has property (†) on Slide 105.]

(ii) $L_2 \triangleq \{w \in \{a,b\}^* \mid w \text{ a palindrome}\}$

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

(iii) $L_3 \triangleq \{a^p \mid p \text{ prime}\}$

[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 (†).]

Using the Pumping Lemma

The Pumping Lemma (Slide 102) 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 $\ell \geq 1$ 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 105.

Slide 106 gives some examples:

(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 $|u_1 v| \leq \ell$ and $|v| \geq 1$. 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 $s = |v| \geq 1$).

(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 $n = 0$ case 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 [Euclid proved that] 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 $|u_1 v| \leq \ell$ and $|v| \geq 1$. Letting $r \triangleq |u_1|$ and $s \triangleq |v|$, so that $|u_2| = p - r - s$, 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 $s = |v| \geq 1$) and $p - s > 2\ell - \ell = \ell \geq 1$ (since $p > 2\ell$ by choice, and $s \leq r + s = |u_1 v| \leq \ell$). Therefore $u_1 v^n u_2 \notin L_3$ when $n = p - s$.

Remark. Unfortunately, the method on Slide 105 cannot 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 102, but which nonetheless, are not regular. Slide 109 gives an example of such an L.

Example of a non-regular language with the pumping lemma property

$$L \triangleq \{c^m a^n b^n \mid m \geq 1 \,\&\, n \geq 0\} \cup \{a^m b^n \mid m, n \geq 0\}$$

satisfies the pumping lemma property on Slide 102 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 5.1.