# Formal Languages and Automata

5 lectures for

**2017-18 Computer Science Tripos**
**Part IA Discrete Mathematics**
by Ian Leslie

## What is this course about?

- Examining the power of an abstract machine
- Domains of discourse: automata and formal languages
- Formalisms to describe languages and automata
- Proving a particular case: relationship between regular languages and finite automata

Perhaps the simplest result about power of a machine. Finite Automata are simply a formalisation of finite state machines you looked at in Digital Electronics.

## A word about formalisms to describe languages

- Classically (i.e. when I was young) this would be done using production-based grammars.
- Here will we use rule induction

Excuse to introduce rule induction now, useful in other things

## Syllabus for this part of the course

- Inductive definitions using rules and proofs by rule induction.
- Regular expressions and pattern matching.
- Finite automata and regular languages: Kleene's theorem.
- The Pumping Lemma.

mathematics needed for computer science

**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, Temporal Logic and Model Checking |

N.B. we do not cover the important topic of context-free grammars, which prior to 2013/14 was part of the CST IA course *Regular Languages and Finite Automata* that has been subsumed into this course.

see course web page for relevant Tripos questions

## Formal Languages

## Alphabets

An **alphabet** is specified by giving a finite set, $\Sigma$, 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.
- $\{S \mid S \subseteq \{0,1,2,3,4,5,6,7,8,9\}\}$, $2^{10}$-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 $\Sigma$ is just an ordered $n$-tuple of elements of $\Sigma$, written without punctuation.

$\Sigma^*$ denotes set of all strings over $\Sigma$ of any finite length.

**Examples:**

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

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

$$\begin{aligned} vu &= raab \\ uu &= abab \\ wv &= cadra \\ uvwuv &= abracadabra \end{aligned}$$

N.B. $(uv)w = uvw = u(vw)$ ( any u,v,w )
$u\epsilon = u = \epsilon u$

The length of a string $u \in \Sigma^*$ is denoted $|u|$.

## 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 $\Sigma$, we call any subset of $\Sigma^*$ a (formal) **language** over the alphabet $\Sigma$.

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

## 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 $U$
  which means that $a$ is in the subset we are defining
- **rules** $\boxed{\dfrac{h_1 \; h_2 \; \cdots \; h_n}{c}}$
  are specified by giving a finite subset $\{h_1, h_2, \dots, h_n\}$ of $U$ (the **hypotheses** of the rule) and an element $c$ of $U$ (the **conclusion** of the rule)
  which means that $c$ is in the subset we are defining if all of $h_1, h_2, \dots, h_n$ are

## 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 rooted tree with vertexes 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.

we'll draw with leaves at top, root at bottom

## Example

$U = \{a, b\}^*$ The universal set.
Axioms and Rules:

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

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

Example derivations:

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

## 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.)

rules or templates?

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

is really a template for a (potentially) infinite set of rules

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

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

we can use Rule Induction 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.

## 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 $u$.

Derivation is a finite (labelled) tree with $u$ at root, axiom at leaves and each vertex the conclusion of a rule whose hypotheses are the children of the vertex.

(We usually draw the trees with the root at the bottom.)

## 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 $c \in S$.

E.G. for the axiom & rules

$$\dfrac{}{\epsilon} \qquad \dfrac{u}{aub} \qquad \dfrac{u}{bua} \qquad \dfrac{u\ v}{uv} \qquad \text{for all } u,v \in \{a,b\}^*$$

the subset

$$\{u \in \{a,b\}^* \mid \#_a(u) = \#_b(u)\}$$

is closed under the axiom & rules.

N.B. for a given set $\mathcal{R}$ of axioms & rules

$$\boxed{\{u \in U \mid \forall S \subseteq U.(S \text{ closed under } \mathcal{R}) \Longrightarrow u \in S\}}$$

is closed under $\mathcal{R}$ (Why?) and so is the smallest such (with respect to subset inclusion, $\subseteq$)

This set contains all items that are in every set that is closed under $\mathcal{R}$

Perhaps better written as

$$\boxed{\bigcap(\forall S \subseteq U.(S \text{ closed under } \mathcal{R}))}$$

is closed under $\mathcal{R}$.

**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 $I \subseteq S$.

"the least subset closed under the axioms & rules"

is sometimes take as the definition of

"inductively defined subset"

## Proof of the Theorem [Page 23 of notes]

### Closure part

- $I$ is closed under each axiom $\dfrac{}{a}$

  Because we can construct a derivation witnessing $a \in I$ ...

  ...which is simply a tree with one node containing $a$

### Closure part (2)

- $I$ is closed under each rule $r = \dfrac{h_1\ h_2\ \dots h_n}{c}$

  Because if $h_1\ h_2 \dots h_n \in I$ ...

  we have $n$ derivations from axioms to each $h_i$ and so ...

  we can just make these the $n$ children to our rule $r$ to form a big tree ...

  which is a derivation witnessing $c \in I$

## Proof of the Theorem

so we have closure under rules & axioms

Now the "least such subset" part

We need to show, for every $S \subseteq U$

$$\boxed{(S \text{ closed under axioms and rules }) \Rightarrow \quad I \subseteq S}$$

That is, $I$ is the least subset, in that any other subset that is closed under the axioms & rules contains $I$.

## Least Subset

So we need to show that every element of $I$ is contained in any set $S \subseteq U$ which is closed under the rules & axioms

Q: How can we characterise an element of $I$?
A: For each element of $I$ there is a derivation that witnesses its membership

So let's do induction on the height of the derivation (i.e. the height of the tree)

## Least Subset - Proof By Induction

$P(n) \triangleq$ "all derivations of height $n$ have their conclusion in $S$"

Need to show:

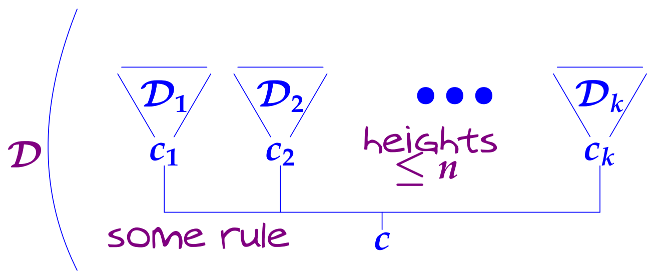
- $P(0)$ (consider these to be single (axiom) node derivations)
- $\forall(k \leq n)\ P(k) \Rightarrow P(n+1)$

since if $P(n)$ is true for all $n$, then all derivations have their conclusion in $S$, and thus every element of $I$ is in $S$.

## Least Subset - Proof By Induction

$P(n) \triangleq$ "all derivations of height $n$ have their conclusion in $S$"

- $P(0)$:
  trivially true since conclusion is an axiom and $S$ is closed under axioms
- $\forall(k \leq n)\ P(k) \Rightarrow P(n+1)$:
  Suppose $\forall(k \leq n)\ P(k)$ and that $\mathcal{D}$ is a derivation of height $n+1$ with, say, conclusion $c$



But the derivations for the $c_i$ all have height $\leq n$. So the $c_i$ are all in $S$ by assumption

and since $S$ is closed under all axioms & rules, $c \in S$

so $\forall(k \leq n)\ P(k) \Rightarrow P(n+1)$

Thus every element in $I$ is in any $S$ that is closed under the axioms & rules that inductively defined $I$.

Thus $I$ is the least subset that is closed under those axioms & rules.

## 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 $\dfrac{h_1\ h_2\ \cdots\ h_n}{c}$

## Example using rule induction

Let $I$ be the subset of $\{a,b\}^*$ inductively defined by the axioms and rules on Slide 17 of the notes.

$$\boxed{\frac{}{\epsilon} \qquad \frac{u}{aub} \qquad \frac{u}{bua} \qquad \frac{u\ v}{uv}}$$

Associated Rule Induction:

- $P(\epsilon)$
- $\forall u \in I\,.\,P(u) \Rightarrow P(aub)$
- $\forall u \in I\,.\,P(u) \Rightarrow P(bua)$
- $\forall u, v \in I\,.\,P(u) \wedge P(v) \Rightarrow P(uv)$

## Example using rule induction

Let $I$ be the subset of $\{a, b\}^*$ inductively defined by the axioms and rules on Slide 17 of the notes.

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 $a$s and $b$s 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.)

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a, b\}^*$ inductively defined by

$$\boxed{\frac{}{a}\,0 \qquad \frac{u}{au}\,1 \qquad \frac{u\ v}{buv}\,2}$$

In this case Rule Induction says:
if (0) $P(a)$
& (1) $\forall u \in I\ .\ P(u) \Rightarrow P(au)$
& (2) $\forall u, v \in I\ .\ P(u) \wedge P(v) \Rightarrow P(buv)$
then $\forall u \in I\ .\ P(u)$

for $\underline{\text{any}}$ predicate $P(u)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a, b\}^*$ inductively defined by

$$\boxed{\frac{}{a}\,0 \qquad \frac{u}{au}\,1 \qquad \frac{u\ v}{buv}\,2}$$

Asked to show

$$u \in I \Rightarrow \#_a(u) > \#_b(u)$$

so do so using Rule Induction with

$$P(u) = \#_a(u) > \#_b(u)$$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a, b\}^*$ inductively defined by

$$\boxed{\frac{}{a}\,0 \qquad \frac{u}{au}\,1 \qquad \frac{u\ v}{buv}\,2}$$

$$P(u) = \#_a(u) > \#_b(u)$$

(0) $P(a)$ holds $(1 > 0)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\dfrac{}{a}\,0 \qquad \dfrac{u}{au}\,1 \qquad \dfrac{u\ v}{buv}\,2}$$

$$P(u) = \#_a(u) > \#_b(u)$$

(1) If $P(u)$, then $\#_a(au) = 1 + \#_a(u)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\dfrac{}{a}\,0 \qquad \dfrac{u}{au}\,1 \qquad \dfrac{u\ v}{buv}\,2}$$

$$P(u) = \#_a(u) > \#_b(u)$$

(1) If $P(u)$, then $\#_a(au) = 1 + \#_a(u)$
$> \#_a(u) > \#_b(u)$ (because $P(u)$)
$= \#_b(au)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\dfrac{}{a}\,0 \qquad \dfrac{u}{au}\,1 \qquad \dfrac{u\ v}{buv}\,2}$$

$$P(u) = \#_a(u) > \#_b(u)$$

(1) If $P(u)$, then $\#_a(au) = 1 + \#_a(u)$
$> \#_a(u) > \#_b(u)$ (because $P(u)$)
$= \#_b(au)$
so $P(au)$ holds as well, and thus $P(u) \Rightarrow P(au)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\dfrac{}{a}\,0 \qquad \dfrac{u}{au}\,1 \qquad \dfrac{u\ v}{buv}\,2}$$

$$P(u) = \#_a(u) > \#_b(u)$$

(2) If $P(u) \wedge P(v)$, then $\#_a(buv) = \#_a(u) + \$\#_a(v)$
$\geq ((\#_b(u) + 1) + (\#_b(v) + 1))$ (why?)
$> \#_b(buv)$
so $P(buv)$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\frac{}{a}\,{\scriptstyle 0} \qquad \frac{u}{au}\,{\scriptstyle 1} \qquad \frac{u\ v}{buv}\,{\scriptstyle 2}}$$

$$P(u) = \#_a(u) > \#_b(u)$$

if (0) $P(a)$ ✓
& (1) $\forall u \in I \,.\, P(u) \Rightarrow P(au)$ ✓
& (2) $\forall u,v \in I \,.\, P(u) \wedge P(v) \Rightarrow P(buv)$ ✓
then $\forall u \in I \,.\, P(u)$
so for all $u \in I$, we have $\#_a(u) > \#_b(u)$ $\square$

## Example [CST 2009, Paper2, Question 5]

$I \subseteq \{a,b\}^*$ inductively defined by

$$\boxed{\frac{}{a}\,{\scriptstyle 0} \qquad \frac{u}{au}\,{\scriptstyle 1} \qquad \frac{u\ v}{buv}\,{\scriptstyle 2}}$$

$$P(u) = \#_a(u) > \#_b(u)$$

although we have $\forall u \in I \,.\, P(u)$

we don't have $\forall u \in \{a,b\}^* \,.\, P(u) \Rightarrow u \in I$

e.g. $P(aab)$ but $aab \notin I$ (Why?)

## Collatz Conjecture

$$f(n) = \begin{cases} 1 & \text{if } n = 0,1 \\ f(n/2) & \text{if } n > 1,\ n \text{ even} \\ f(3n+1) & \text{if } n > 1,\ n \text{ odd} \end{cases}$$

Does this define a <u>total</u> function $f : \mathbb{N} \to \mathbb{N}$?

(nobody knows)

Can reformulate as a problem about inductively defined subsets...

## Collatz Conjecture

$$f(n) = \begin{cases} 1 & \text{if } n = 0,1 \\ f(n/2) & \text{if } n > 1,\ n \text{ even} \\ f(3n+1) & \text{if } n > 1,\ n \text{ odd} \end{cases}$$

Is the subset $I \subseteq \mathbb{N}$ inductively defined by

$$\boxed{\frac{}{0} \qquad \frac{}{1} \qquad \frac{k}{2k} \qquad \frac{6k+4}{2k+1} \quad (k \geq 1)}$$

equal to the whole of $\mathbb{N}$?

## Regular Expressions

## Formal languages

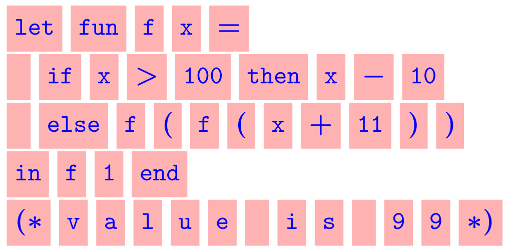
An extensional view of what constitutes a formal language is that it is completely determined by the set of 'words in the dictionary':

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

**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:

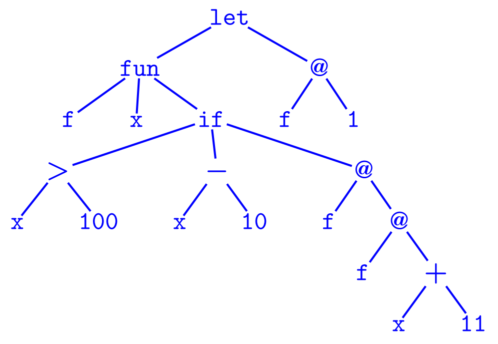


```
let fun f x =
  if x > 100 then x − 10
  else f ( f ( x + 11 ) )
in f 1 end
(* v a l u e   i s   9 9 *)
```

**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 $0$-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 42:



## Regular Expressions

A regular expression defines a pattern of symbols (and thus a language).

Important to distinguish between the language a particular regular expression defines and the set of possible regular expressions.

We about to look at the second of these.

## Regular expressions (concrete syntax)

over a given alphabet $\Sigma$.

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

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

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 $a, b \in \Sigma$)

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

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

## Regular expressions (abstract syntax)

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

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

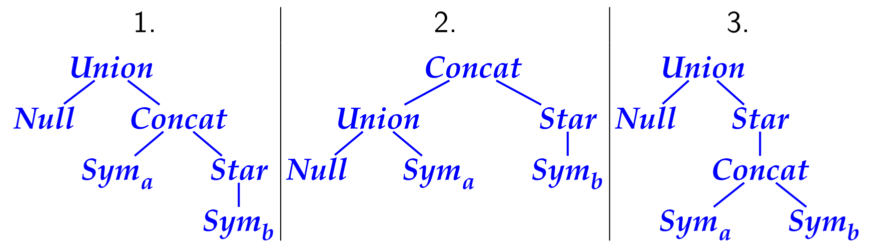
## Regular expressions (abstract syntax)

The 'signature' for regular expression abstract syntax trees (over an alphabet $\Sigma$) 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 $\Sigma$)

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



(*cf.* examples a few slides previous)

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 $\Sigma$, 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:

$$\begin{aligned} \epsilon|(a(b^*)) &\sim Union(Null, Concat(Sym_a, Star(Sym_b))) \\ \epsilon|ab^* &\sim Union(Null, Concat(Sym_a, Star(Sym_b))) \\ \epsilon|ab^* &\sim Concat(Union(Null, Sym_a), Star(Sym_b)) \end{aligned}$$

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.)

## Operator precedence for regular expressions

$\boxed{\text{Star > Concat > Union}}$

So

$\varepsilon|ab^*$ stands for $\varepsilon|(a(b^*))$

Union (Null, Concat (Sym$_a$, Star (Sym$_b$)))

## Associativity for regular expressions

$\boxed{\text{Concat \& Union are \underline{left} associative}}$

So

$abc$ stands for $(ab)c$

$a|b|c$ stands for $(a|b)|c$

From now on, we will rely on operator precedence (& associativity) conventions in the concrete syntax of regular expressions to allow us to map unambiguously to their abstract syntax

associativity less important (in some sense) than precedence because the meaning (semantics) of concatenation and union is always associative but not true of all operators, e.g. division

so $abc$ has the same abstract syntax as $(ab)c$, but different abstract syntax from $a(bc)$, but all of these have the same semantics.

## Matching

Each regular expression $r$ over an alphabet $\Sigma$ 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 $\epsilon$ iff $u$ is the null string $\varepsilon$
- 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

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

abstract syntax trees

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

rules:

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

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

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

## Questions Computer Scientists ask

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

in other words, decides, for any $r$, whether $u \in L(r)$

An algorithm? what's an algorithm? I mean what is it in a mathematical sense?

leads us to define automata which "execute algorithms"
next chunk of the course...

## Questions Computer Scientists ask

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

Yes

Yes because there are convenient notations like $[a-z]$ to mean $a|b|c\dots|z$ and complement, $\sim r$, which is defined to match all strings that $r$ does not. Look at the unix utility grep.

## Questions Computer Scientists ask

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

Yes and No

Yes because there are convenient notations like $[a - z]$ to mean $a|b|c \dots |z$ and complement, $\sim r$, which is defined to match all strings that $r$ does not. Look at the unix utility grep.

No because such conveniences don't allow us to define languages we can't already define

Why not include them in our basic definition??

Because they give us more rules to analyse!

## Questions Computer Scientists ask

(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?

We will answer this when we answer (a).

## Questions Computer Scientists ask

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

Pretty clearly no.

in fact even simple languages like $a^n b^n, \forall n \in \mathbb{N}$ or well-bracketed arithmetic expressions are not regular

we will derive and use the Pumping Lemma to show this

## 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 $\Sigma^*$) of the form $L(r)$ for some $r$?

## Finite Automata

We are about to describe some different types of finite automata.

The game plan is as follows:

- define (non-deterministic) finite automata in general
- define deterministic finite automata (as a special case)
- define non-deterministic finite automata with $\varepsilon$-transitions
- show that from any non-deterministic finite automaton with $\varepsilon$-transitions we can mechanically produce an equivalent deterministic finite automaton

Why?

- we are claiming that a deterministic finite automata (DFA) is an embodiment of an algorithm
- non-deterministic finite automata with $\varepsilon$-transitions (NFA$^{\varepsilon}$'s) map on to our problem (matching regular expressions) more naturally . . .
- . . . so we will produced the NFA$^{\varepsilon}$'s we want and then rely on the fact that for each there is an equivalent DFA.

## Example of a finite automaton

$$M \triangleq \begin{tikzcd} {} \arrow[r] & q_0 \arrow[r, "a"] \arrow[loop below, "b"] & q_1 \arrow[l, bend right, "b"'] \arrow[r, "a"] & q_2 \arrow[ll, bend left, "b"] \arrow[r, "a"] & \boxed{q_3} \arrow[loop above, "a"] \arrow[loop below, "b"] \end{tikzcd}$$

- 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): $q_3$

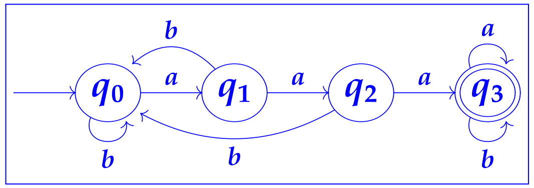
## Language accepted by a finite automaton $M$

- 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 $q = q'$.)

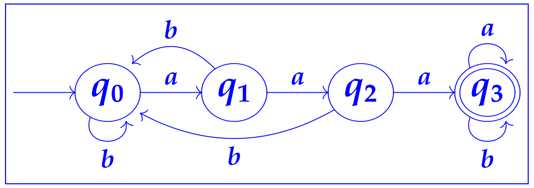
## Example of an accepted language

$M \triangleq$ 

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 $a$'s)

## 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**)
- $\Sigma$ is a finite set (the alphabet of **input symbols**)
- $\Delta$ 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 $(q, a, q') \in \Delta$.

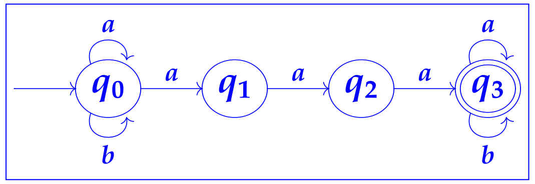
Why do we say this is non-deterministic?

$\Delta$, the transition relation specifies a set of next states for a given current state and given input symbol.

That set might have 0, 1 or more elements.

## 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 our first automaton, namely $\{u \in \{a, b\}^* \mid u \text{ contains three consecutive } a\text{'s}\}$.

So we define a deterministic finite automata so that $\Delta$ is restricted to specify exactly one next state for any given state and input symbol

we do this by saying the relation $\Delta$ has to be a function $\delta$ from $Q \times \Sigma$ to $Q$

## 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 $\delta$ and call the **next-state function**.

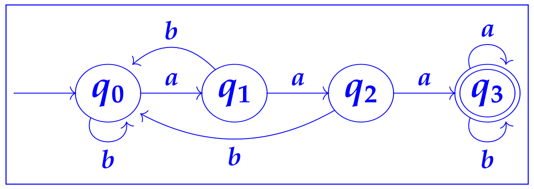
Thus for each (state, input symbol)-pair $(q, a)$, $\delta(q, a)$ is the unique state that can be reached from $q$ by a transition labelled $a$:

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

## Example of a DFA...

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

$M \triangleq$



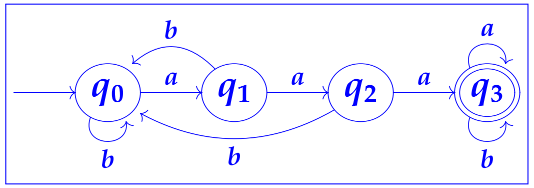
next-state function:

| $\delta$ | $a$ | $b$ |
|---|---|---|
| $q_0$ | $q_1$ | $q_0$ |
| $q_1$ | $q_2$ | $q_0$ |
| $q_2$ | $q_3$ | $q_0$ |
| $q_3$ | $q_3$ | $q_3$ |

## but this is an NFA

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

$M \triangleq$



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

so alphabet matters!

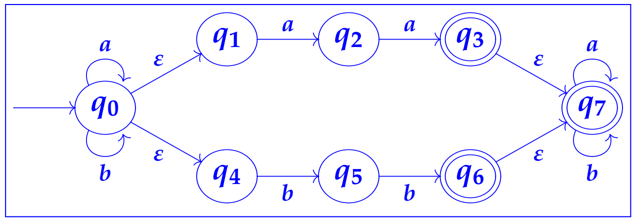
Now let's make things a bit more interesting (well complicated) ...

We are going to introduce a new form of transition, an $\varepsilon$-transition which allows us to move from one state to another without reading a symbol.

These (in general) introduce non-determinism all by themselves.

An **NFA with $\varepsilon$-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 **$\varepsilon$-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$^\varepsilon$

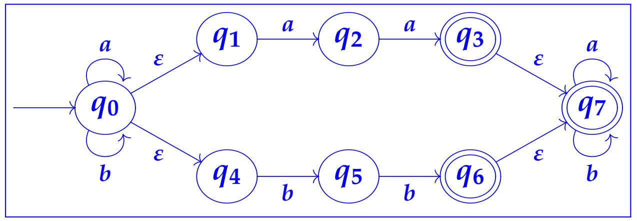
$M = (Q, \Sigma, \Delta, s, F, T)$

- Look at paths in the transition graph (including $\varepsilon$-transitions) from start state to *some* accepting state.
- Each such path gives a string in $\Sigma^*$, namely the string of non-$\varepsilon$ 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 $q \overset{u}{\Rightarrow} q'$ to mean that there is a path in $M$ from state $q$ to state $q'$ whose non-$\varepsilon$ labels form the string $u \in \Sigma^*$.

An **NFA with $\varepsilon$-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 **$\varepsilon$-transition relation**.

**Example**:



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

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

## Sets of Languages Accepted by Finite Automata

- every DFA is an NFA (with transition mapping $\Delta$ being a next-state function $\delta$)
- every NFA is an NFA$^\varepsilon$ (with empty $\varepsilon$-transition relation)

clearly

$$L(\mathbb{DFA}) \subseteq L(\mathbb{NFA}) \subseteq L(\mathbb{NFA}^\epsilon)$$

But

$$L(\mathbb{DFA}) \subset L(\mathbb{NFA}) \subset L(\mathbb{NFA}^\epsilon)???$$

## NFA$^\varepsilon$ accepts if there exists a path...

DFA: path is determined one symbol at a time

Let Q be the states of some NFA$^\varepsilon$. What if we thought, one symbol at a time, about the states we could be in, or more precisely the subset of Q containing the states we could be in

Then we could construct a new DFA whose states were taken from the powerset of Q from the NFA$^\varepsilon$

## Subset Construction

Given an NFA$^\varepsilon$ $M$ with states $Q$ construct a DFA $PM$ whose states are subsets of the states of $M$

the start state in $PM$ would be a set containing the start state of $M$ together with any states that can be reached by $\varepsilon$-transitions from that state.

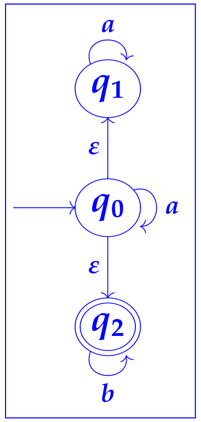
accepting states in $PM$ would be any subset containing an accepting state of $M$

alphabet is the same as the alphabet of $M$

That just leaves $\delta$

## Example of the subset construction

$M$



next-state function for $PM$

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

## A word about $\emptyset$ in the subset construction

Potential for confusion

- The DFA has a state which corresponds to the empty set of states in the NFA$^\varepsilon$ which we have designated as $\emptyset$.
- Once you enter this state we get stuck in it. Why?
- Could rewrite (next slide)

| DFA State | subset of NFA$^\varepsilon$ | $a$ | $b$ |
|---|---|---|---|
| $S_1$ | $\emptyset$ | $S_1$ | $S_1$ |
| $S_2$ | $\{q_0\}$ | $S_8$ | $S_4$ |
| $S_3$ | $\{q_1\}$ | $S_3$ | $S_1$ |
| $S_4$ | $\{q_2\}$ | $S_2$ | $S_4$ |
| $S_5$ | $\{q_0, q_1\}$ | $S_8$ | $S_4$ |
| $S_6$ | $\{q_0, q_2\}$ | $S_8$ | $S_4$ |
| $S_7$ | $\{q_1, q_2\}$ | $S_3$ | $S_4$ |
| $S_8$ | $\{q_0, q_1, q_2\}$ | $S_8$ | $S_4$ |

Noting that $S_8$ is the start state (why?) we could eliminate states that can't be reached (i.e. $S_2$, $S_5$, $S_6$ and $S_7$; and thence $S_3$) if we cared. Here we don't. (Care that is).

**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 $\Sigma$ as for $M$
- next-state function maps each $(S, a) \in \mathcal{P}(Q) \times \Sigma$ to $\delta(S, a) \triangleq \{q' \in Q \mid \exists q \in S.\ q \overset{a}{\Rightarrow} q' \text{ in } M\}$
- start state is $s' \triangleq \{q' \in Q \mid s \overset{\varepsilon}{\Rightarrow} q'\}$
- subset of accepting sates 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 $L(PM) \subseteq L(M)$.

Consider a string $a_1 a_2 \ldots a_n \in L(M)$, i.e. is accepted by our NFA$^\varepsilon$ $M$

Then we have

$$\begin{tikzcd}
s \arrow[r, Rightarrow, "a_1"] \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] & q_1 \arrow[r, Rightarrow, "a_2"] \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] & \ldots \arrow[r, Rightarrow, "a_n"] & q_n \arrow[d, phantom, "\rotatebox{-90}{$\in$}"] & \in F \quad \text{in } M \\
S' \arrow[r, "a_1"] & S_1 \arrow[r, "a_2"] & \ldots \arrow[r, "a_n"] & S_n & \in F' \quad \text{in } PM
\end{tikzcd}$$

so $a_1 a_2 \ldots a_n \in L(PM)$

so $L(M) \subseteq L(PM)$

Consider a string $a_1 a_2 \ldots a_n \in L(PM)$, i.e. is accepted by our DFA $PM$

Then we have

$$\begin{tikzcd}
S' \arrow[r, "a_1"] & S_1 \arrow[r, "a_2"] & \ldots & S_{n-1} \arrow[r, "a_n"] & S_n \in F' \quad \text{in } PM \\
q_0 \arrow[u, phantom, "\rotatebox{90}{$\in$}"] \arrow[r, Rightarrow, "a_1"] & q_1 \arrow[u, phantom, "\rotatebox{90}{$\in$}"] \arrow[r, Rightarrow, "a_2"] & \ldots & q_{n-1} \arrow[u, phantom, "\rotatebox{90}{$\in$}"] \arrow[r, Rightarrow, "a_n"] & q_n \in F \quad \text{in } M \arrow[u, phantom, "\rotatebox{90}{$\in$}"] \\
s \arrow[u, Rightarrow, "\varepsilon"'] & & & &
\end{tikzcd}$$

so $a_1 a_2 \ldots a_n \in L(M)$

so $L(PM) \subseteq L(M)$

So we have shown

$$L(M) \subseteq L(PM) \text{ and } L(PM) \subseteq L(M)$$

so that

$$L(M) = L(PM)$$

where $PM$ is specified by $M$ through subset construction.

Thus for every NFA$^\varepsilon$ there is an <u>equivalent</u> DFA

**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 $\Sigma$ as for $M$
- next-state function maps each $(S, a) \in \mathcal{P}(Q) \times \Sigma$ to $\delta(S, a) \triangleq \{q' \in Q \mid \exists q \in S.\, q \overset{a}{\Rightarrow} q' \text{ in } M\}$
- start state is $s' \triangleq \{q' \in Q \mid s \overset{\varepsilon}{\Rightarrow} q'\}$
- subset of accepting sates 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 $L(PM) \subseteq L(M)$.

At this point we should think of

- the set of all language $\{L(r)\}$ defined by a some regular expression $r$, each language being the set of strings which match some regular expression $r$
- the set of all languages $\{L(M)\}$ accepted by some deterministic finite automaton $M$

## Kleene's Theorem

## 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 the form $L(r)$ for some regular expression $r$.

The first part requires us to demonstrate that for any regular expression $r$, we can construct a DFA, $M$ with $L(M) = L(r)$

We will do this by demonstrating that for any $r$ we can construct a NFA$^\varepsilon$ $M'$ with $L(M') = L(r)$ and rely on the subset construction theorem to give us the DFA $M$.

We consider each axiom and rule that define regular expressions

## Kleene's Theorem Part a (The Fun Part)

For any regular expression $r$ we can build an NFA$^\varepsilon$ $M$ such that $L(r) = L(M)$

We will work on induction on the depth of abstract syntax trees

## Recall: Regular expressions (abstract syntax)

(concrete syntax)

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

- binary operators $Union$ and $Concat$ $\quad r_1|r_2 \quad r_1r_2$
- unary operator $Star$ $\quad r^*$
- nullary operators (constants) $Null$, $Empty$ and $Sym_a$ (one for each $a \in \Sigma$). $\quad \epsilon \quad \emptyset \quad a$

(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

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

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 $r_1r_2$:** given NFA$^\varepsilon$s $M_1$ and $M_2$, construct an NFA$^\varepsilon$ $Concat(M_1, M_2)$ satisfying

$$\boxed{L(Concat(M_1, M_2)) = \{u_1u_2 \mid u_1 \in L(M_1) \;\&\; u_2 \in L(M_2)\}}$$

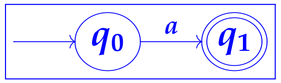
Thus $L(r_1r_2) = L(Concat(M_1, M_2))$ when $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$.

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

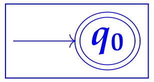
$$\boxed{L(Star(M)) = \{u_1u_2 \dots u_n \mid n \geq 0 \text{ and each } u_i \in L(M)\}}$$

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

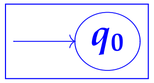
## NFAs for regular expressions $a$, $\epsilon$, $\emptyset$



just accepts the one-symbol string $a$

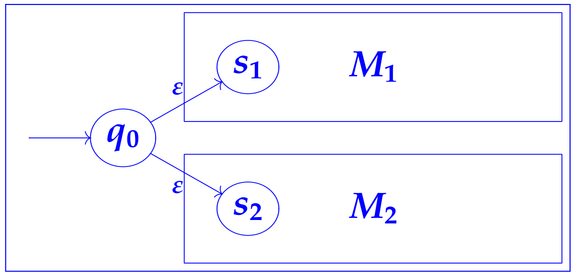


just accepts the null string, $\varepsilon$



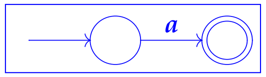
accepts no strings

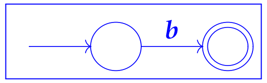
## $Union(M_1, M_2)$

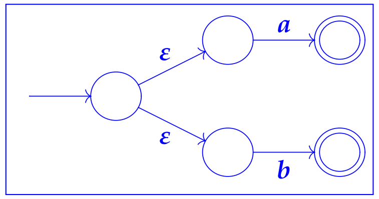


accepting states = union of accepting states of $M_1$ and $M_2$

For example,

if $M_a =$ 

and $M_b =$ 

then $Union(M_a, M_b) =$ 

In what follows, whenever we have to deal with two machines, say $M_1$ and $M_2$ together, we assume that their states are disjoint.

If they were not, we could just rename the states of one machine to make this so.

Also assume that for $r_1$ and $r_2$ there are machines $M_1$ and $M_2$ such that $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$

## Construction for $Union(r_1, r_2)$

Assume there are two machines $M_1$ and $M_2$ with $L(r_1) = L(M_1)$ and $L(r_2) = L(M_2)$

States of new machine $M = Union(M_1, M_2)$ are all the states in $M_1$ and all the states in $M_2$ together with a new start state with $\varepsilon$-transitions to each of the (old) start states of $M_1$ and $M_2$.

Accept states of $M$ are the all accept states in $M_1$ and all accept states in $M_2$.

The transitions of $M$ are all transitions in $M_1$ and $M_2$ along with the two $\varepsilon$-transitions from the new start state

$M$ accepts any strings that $M_1$ accepts:

if $u \in L(M_1)$ then $s_1 \overset{u}{\Rightarrow} q_1$ where $s_1$ is start state and $q_1$ an accept state of $M_1$ respectively.

But then in $M$, $s \overset{u}{\Rightarrow} q_1$, where $s$ is our new start state since $s \xrightarrow{\varepsilon} s_1$.

so $u \in L(M)$. Similar argument for $M$ accepting any string that $M_2$ accepts

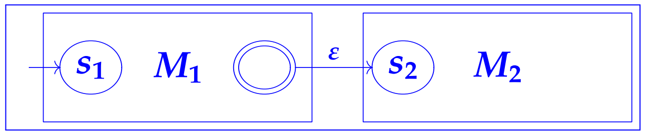
so $(L(M_1) \cup L(M_2)) \subseteq L(Union(M_1, M_2))$

Can $M$ accept anything more?

The only way "out of" $s$, the start state of $M$, is either to the start state of $M_1$ or the start state of $M_2$

So no, $L(M) = (L(M_1) \cup L(M_2))$

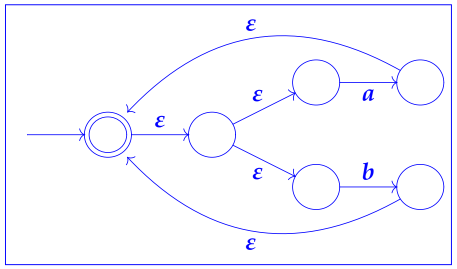
## $Concat(M_1, M_2)$



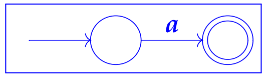
accepting states are those of $M_2$

For example,

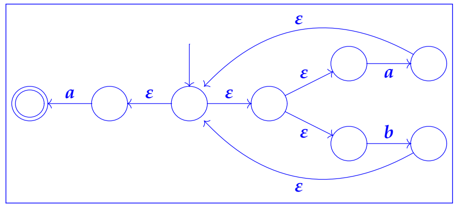
if $M_1 =$



and $M_2 =$



then $Concat(M_1, M_2) =$

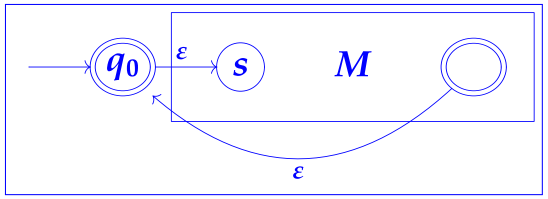


## Construction for $M = Concat(M_1, M_2)$

Make an $\varepsilon$-transition from every accept state in $M_1$ to the start state of $M_2$.

Start state of $M$ is the start state of $M_1$; accept states of $M$ are the accept states of $M_2$

## $Star(M)$

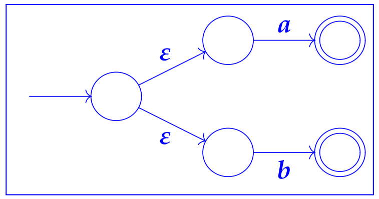


the only accepting state of $Star(M)$ is $q_0$

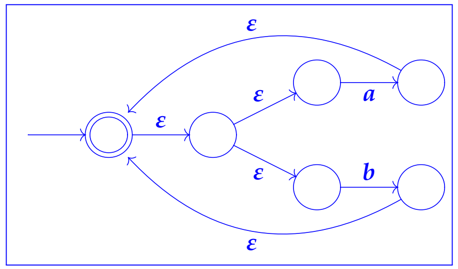
(N.B. doing without $q_0$ by just looping back to $s$ and making that accepting won't work – see exercises)

For example,

if $M =$



then $Star(M) =$



Construction for $Star(r_1)$, $M = Star(M_1)$

Create a new state, say $s$ which will be the start state, and the only accepting state of $M$.

The transitions of $M$ are all the transitions of $M_1$ together with an $\varepsilon$-transition from $s$ to the (old) start state of $M_1$ and $\varepsilon$-transitions from every (old) accepting state of $M_1$ to $s$.

Clearly, $M$ accepts $\varepsilon$ since $s$, the start state, is also an accepting state

nonempty strings accepted by $M$ have to be formed of components, each of which is accepted by $M_1$

so $L(M) = L(r_1^*)$

(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

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

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

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

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

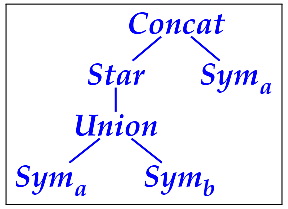
$$\boxed{L(Star(M)) = \{u_1 u_2 \ldots u_n \mid n \geq 0 \text{ and each } u_i \in L(M)\}}$$

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

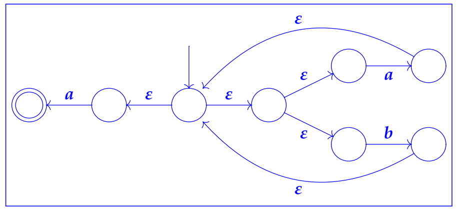
## Example

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

whose abstract syntax tree is



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



## 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 $\Sigma^*$) of the form $L(r)$ for some $r$?

## Decidability of matching

We now have a positive answer to question (a). 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 ) 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 $PM$.)

Exponential Blow-up

if $NFA^\varepsilon$ $M$ has $n$ states then the DFA made by subset construction, $PM$ has $2^n$ states, since its states are the members of the powerset of $M$.

Minimisation of states in $PM$ by:

- removing all states which are not reachable (by any string) from the start state.
- merge all compatible states. Two states are compatible if (i) they are both accepting or both non-accepting; and (ii) their transition functions are the same.
- Update transition functions to take account of merged states. Repeat.

## 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.**

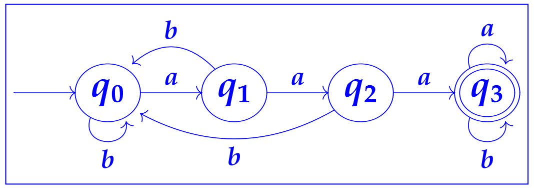
(a) For any regular expression $r$, the set $L(r)$ of strings matching $r$ is a regular language.

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

## Example of a regular language

Recall the example DFA we used earlier:

$M \triangleq$

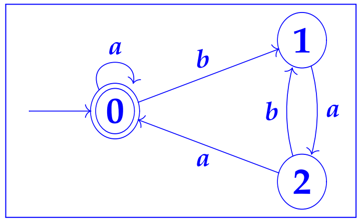


$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "a"] \arrow[loop below, "b"] & q_1 \arrow[l, bend right, "b"'] \arrow[r, "a"] & q_2 \arrow[ll, bend left, "b"] \arrow[r, "a"] & q_3 \arrow[loop above, "a"] \arrow[loop below, "b"] \end{tikzcd}$$

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).

**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

$L(r^S_{q,q'}) = \{u \in \Sigma^* \mid q \xrightarrow{u}{}^* q'$ in $M$ 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 $\emptyset$).

Prove this Lemma by induction on # of elements in $S$
Also take care to examine case where $q = q'$ !

Base case $S = \emptyset$

Given states $q, q' \in M$, if

$$q \xrightarrow{a} q'$$

holds for just $a = a_1, a_2, \ldots, a_k$ then can define

$$r^{\emptyset}_{q,q'} \triangleq \begin{cases} a = a_1|a_2|\ldots|a_k & \text{if } q \neq q' \\ a = a_1|a_2|\ldots|a_k|\epsilon & \text{if } q = q' \end{cases}$$

Induction Step:

- $S$ has $n+1$ elements.
- pick some $q_0 \in S$
- consider $S^- = S \setminus \{q_0\}$ ($S$ without the state $q_0$)
- can apply induction hypoth to $S^-$ since $S^-$ has $n$ elements

Can we express $r^S_{q,q'}$ in terms of things only depending on $S^-$?

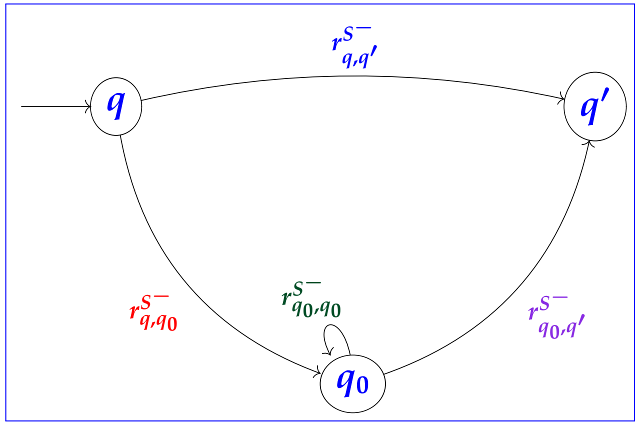
## What's in $r^{S}_{q,q'}$ ?

- we might be able to get from $q$ to $q'$ through $S$ avoiding $q_0$, and
- we might be able to get from $q$ to $q_0$, then from $q_0$ back to itself an arbitrary number of times, then to $q'$

For the first of these we have $r^{S-}_{q,q'}$ by hypothesis. (If there is no path, this will be $\emptyset$)

For the second we have $r^{S-}_{q,q_0}\ [r^{S-}_{q_0,q_0}]^*\ r^{S-}_{q_0,q'}$

$$r^{S}_{q,q'} = r^{S-}_{q,q'} \mid (r^{S-}_{q,q_0}[r^{S-}_{q_0,q_0}]^* r^{S-}_{q_0,q'})$$



all transitions in $S^-$

$q_0$ excluded from $S^-$

$q$ and $q'$ can be in or out of $S^-$

## An Example

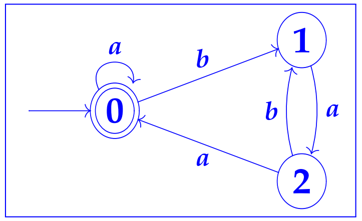
Demonstrates don't always have to follow induction to bitter end (but when in doubt...)

Construction works backwards to the induction; we start with all the states and remove one at a time.

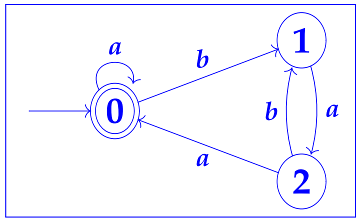
We get to choose the state to remove in each step.

Strategy: choose a state that disconnects the automaton as much as possible

$M \triangleq$



Looking for $r^{\{0,1,2\}}_{0,0}$

$M \triangleq$ 

Looking for $r^{\{0,1,2\}}_{0,0}$

By direct inspection we have:

| $r^{\{0\}}_{i,j}$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | | | |
| 1 | $\emptyset$ | $\varepsilon$ | $a$ |
| 2 | $aa^*$ | $a^*b$ | $\varepsilon$ |

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

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

## We want $r^{\{0,1,2\}}_{0,0}$

## Remove 1 from {0, 1, 2}

$$\begin{aligned} r^{\{0,1,2\}}_{0,0} &\triangleq r^{\{0,2\}}_{0,0} \mid (r^{\{0,2\}}_{0,1}\ [r^{\{0,2\}}_{1,1}]^*\ r^{\{0,2\}}_{1,0}) \\ &= a^* \mid (a^*b\ [r^{\{0,2\}}_{1,1}]^*\ r^{\{0,2\}}_{1,0,}) \end{aligned}$$

## We want $r^{\{0,1,2\}}_{0,0}$

## Remove 2 from {0, 2}

$$\begin{aligned} r^{\{0,1,2\}}_{0,0} &\triangleq r^{\{0,2\}}_{0,0} \mid (r^{\{0,2\}}_{0,1}\ [r^{\{0,2\}}_{1,1}]^*\ r^{\{0,2\}}_{1,0}) \\ &= a^* \mid (a^*b\ [r^{\{0,2\}}_{1,1}]^*\ r^{\{0,2\}}_{1,0,}) \end{aligned}$$

$$\begin{aligned} r^{\{0,2\}}_{1,1} &\triangleq r^{\{0\}}_{1,1} \mid (r^{\{0\}}_{0,2}\ [r^{\{0\}}_{2,2}]^*\ r^{\{0\}}_{2,1}) \\ &= \varepsilon \mid (a\ [\varepsilon]^*\ a^*b) \\ &= \varepsilon \mid (aa^*b) \end{aligned}$$

## We want $r^{\{0,1,2\}}_{0,0}$

## Remove 2 from {0, 2}

$$\begin{aligned} r^{\{0,1,2\}}_{0,0} &\triangleq r^{\{0,2\}}_{0,0} \mid (r^{\{0,2\}}_{0,1}\ [r^{\{0,2\}}_{1,1}]^*\ r^{\{0,2\}}_{1,0}) \\ &= a^* \mid (a^*b\ [\varepsilon|(aa^*b)]^*\ r^{\{0,2\}}_{1,0}) \end{aligned}$$

$$\begin{aligned} r^{\{0,2\}}_{1,1} &\triangleq r^{\{0\}}_{1,1} \mid (r^{\{0\}}_{0,2}\ [r^{\{0\}}_{2,2}]^*\ r^{\{0\}}_{2,1}) \\ &= \varepsilon \mid (a\ [\varepsilon]^*\ a^*b) \\ &= \varepsilon \mid (aa^*b) \end{aligned}$$

## We want $r_{0,0}^{\{0,1,2\}}$

Remove 2 from {0, 2}

$$\begin{array}{rclllll} r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\ & = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & r_{1,0}^{\{0,2\}}) \end{array}$$

## We want $r_{0,0}^{\{0,1,2\}}$

Remove 2 from {0, 2}

$$\begin{array}{rclllll} r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\ & = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & r_{1,0}^{\{0,2\}}) \\ \\ r_{1,0}^{\{0,2\}} & \triangleq & r_{1,0}^{\{0\}} & | & (r_{1,2}^{\{0\}} & [r_{2,2}^{\{0\}}]^* & r_{2,0}^{\{0\}}) \end{array}$$

## We want $r_{0,0}^{\{0,1,2\}}$

Remove 2 from {0, 2}

$$\begin{array}{rclllll} r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\ & = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & r_{1,0}^{\{0,2\}}) \\ \\ r_{1,0}^{\{0,2\}} & \triangleq & r_{1,0}^{\{0\}} & | & (r_{1,2}^{\{0\}} & [r_{2,2}^{\{0\}}]^* & r_{2,0}^{\{0\}}) \\ & = & \emptyset & | & a^* & (\epsilon)^* & aa^* \end{array}$$

## We want $r_{0,0}^{\{0,1,2\}}$

Remove 2 from {0, 2}

$$\begin{array}{rclllll} r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\ & = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & r_{1,0}^{\{0,2\}}) \\ \\ r_{1,0}^{\{0,2\}} & \triangleq & r_{1,0}^{\{0\}} & | & (r_{1,2}^{\{0\}} & [r_{2,2}^{\{0\}}]^* & r_{2,0}^{\{0\}}) \\ & = & \emptyset & | & a^* & (\epsilon)^* & aa^* \\ & = & aaa^* & & & & \end{array}$$

We want $r_{0,0}^{\{0,1,2\}}$

Remove 2 from {0, 2}

$$\begin{array}{llllll}
r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\
& = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & aaa^*) \\
r_{1,0}^{\{0,2\}} & \triangleq & r_{1,0}^{\{0\}} & | & (r_{1,2}^{\{0\}} & [r_{2,2}^{\{0\}}]^* & r_{2,0}^{\{0\}}) \\
& = & \emptyset & | & a^* & (\epsilon)^* & aa^* \\
& = & aaa^* & & & &
\end{array}$$

We want $r_{0,0}^{\{0,1,2\}}$

$$\begin{array}{llllll}
r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\
& = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & aaa^*)
\end{array}$$

We want $r_{0,0}^{\{0,1,2\}}$

$$\begin{array}{llllll}
r_{0,0}^{\{0,1,2\}} & \triangleq & r_{0,0}^{\{0,2\}} & | & (r_{0,1}^{\{0,2\}} & [r_{1,1}^{\{0,2\}}]^* & r_{1,0}^{\{0,2\}}) \\
& = & a^* & | & (a^*b & [\varepsilon|(aa^*b)]^* & aaa^*)
\end{array}$$

Which might have a simpler form...

## 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 $\Sigma^*$) of the form $L(r)$ for some $r$?

## $Not(M)$

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 $M$:

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

So regular languages are closed under complementation:

- given a regular expression $r$
- build DFA $M$ such that $L(M) = L(r)$ (Kleene (a))
- build $Not(M)$ from $M$ (just defined)
- find $\sim r$ such that $L(\sim r) = L(Not(M))$ (Kleene (b))

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

## Regular languages are closed under intersection

**Theorem.** If $L_1$ and $L_2$ are a regular languages over an alphabet $\Sigma$, 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 $L_1 \cap L_2 = L(Not(PM))$, $PM$ subset-constructed from $M$, 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 $\Sigma^*$) 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 DFA $M$, whether or not it accepts *any string at all*.

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.

That gives us our answer to question (c) (which is yes).

Now onto the last of our questions...

## 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 $\Sigma^*$) of the form $L(r)$ for some $r$?

## Examples of languages that are not regular

- The set of strings over $\{(,),a,b,\dots,z\}$ in which the parentheses '$($' and '$)$' occur well-nested.
- The set of strings over $\{a,b,\dots,z\}$ 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.)

Note similarity to construction in Kleene (B)

Suppose $L = L(M)$ for a DFA $M = (Q, \Sigma, \delta, s, F)$. Taking $\ell$ 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

$$\begin{tikzcd} s = q_0 \arrow[r, "u_1\,*"] & q_i = q_j \arrow[loop above, "v"] \arrow[r, "u_2\,*"] & q_n \in F \end{tikzcd}$$

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

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

## Examples

None of the following three languages are regular:

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

$\boxed{L_1 = \{a^n b^n \mid n \geq 0\}}$

For each $\ell \geq 1$, take $w = a^\ell b^\ell \in L_1$

If $w = u_1 v u_2$ with $|u_1 v| \leq \ell$ & $|v| \geq 1$, then for some $r$ and $s$:

- $u_1 = a^r$
- $v = a^s$, with $r + s \leq \ell$ and $s \geq 1$
- $u_2 = a^{l-r-s} b^\ell$

so $u_1 v^0 u_2 = \quad a^r \; \epsilon \; a^{\ell-r-s} b^\ell = \quad a^{\ell-s} b^\ell$

But $a^{\ell-s} b^\ell \notin L_1$ , so, by the Pumping Lemma, $L_1$ is not a regular language

## 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 (†).]

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

$\boxed{L_3 = \{a^p \mid p \text{ prime}\}}$

For each $\ell \geq 1$ let $w = a^p \in L_3$, $p$ prime & $p > 2\ell$

If $w = u_1 v u_2$ with $|u_1 v| \leq \ell$ & $|v| \geq 1$ ...

then $u_1 = a^r \quad v = a^s \quad u_2 = a^{p-r-s}$

with $s \geq 1$ & $r + s \leq \ell$

so $u_1 v^{p-s} u_2 = \quad a^r \; a^{s(p-s)} \; a^{p-r-s} = \quad a^{(p-s)(s+1)}$

But $s \geq 1 \Rightarrow s + 1 \geq 2$
and $(p - s) > (2\ell - \ell) \geq 1 \Rightarrow (p - s) \geq 2$

so $a^{(p-s)(s+1)} \notin L_3$

## 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 (†).]

(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 (†).]

Pumping Lemma property is necessary
for a language to be regular

It is not sufficient

## 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 with $\ell = 1$.

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

But $L$ is not regular – see Exercise 5.1.

$L$ is not regular: (sketch)

If $L$ is regular there is a DFA $M$ with $L = L(M)$. Let's build a new machine, $M'$ from it.

Take a $c$ transition from the start state of $M$. Make the state you reach the start state of $M'$.

Delete all transitions involving $c$ (and remove $c$ from the alphabet). But don't remove any states and keep the same accept states.

What language does $M'$ recognise?

## The way ahead, in THEORY

- What does is mean for a function to be computable?
  [ IB Computation Theory ]
- Are some computational tasks intrinsiclaly unfeasible?
  [ IB ComplexityTheory ]
- How do we specify and reason about program behaviour?
  [ IB Logic and Proof,
  IB Semantics of PLs ]

## The way ahead, in FORMAL LANGUAGE

- Are there other useful language classes?
- Are there other useful automata classes that have a correspondence to them?
- What if we ask the same questions about them that we asked about regular languages?