

Models of Computation

Week 1:

From regular patterns to DFAs

DFA = Deterministic Finite Automaton
= Finite state machine (Q, Σ, δ)
+ initial state q_0 \diamond
+ set F of accepting states
 \odot (possibly \diamond)

Steve Vickers

Module organization

Lecture (2hr) 9.00 Mon

- handouts (also online + spare copies in School library)

- inc. exercises

Exercise class

2.00 Tue

- in 4 tutor groups

Web page:

www.cs.bham.ac.uk/~sjv/teaching/models

Quickies - unassessed

Classwork - assessed
(hand in at ex. class)

Homework - assessed
(due in by following Mon)

Stretchers - bonus
(due in by following Mon)

Module content

What is computation?

- without mentioning programming languages.

finite state machines

pushdown automata

Turing machines
1930s

← simple but restricted.
Give good solutions to some computational problems

← complicated but general

States and commands

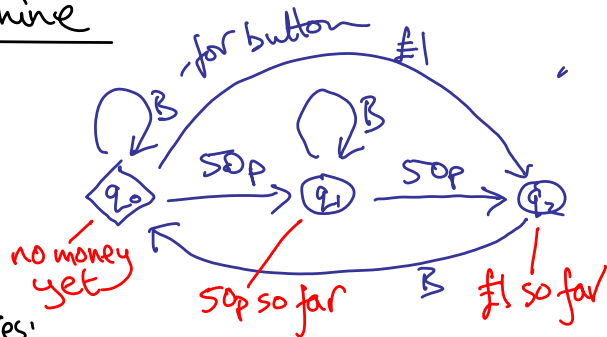
static dynamic-change
A state of a computer?

- describes how it is at a given moment
- no change
- no dynamics

A command changes the state

e.g. Drinks machine

GINGER BEER!
Delicious, only £1
Money SOP, £1
Push button B
for drink

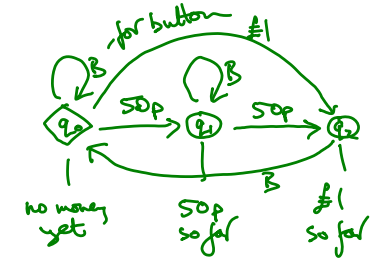


Could have better features:

- Cancel & get money back?
 - Reject surplus money?
 - Keep account of how many drinks paid for?
 - Give change?
- see exercises
- More complex than "finite states"

Transition tables

	commands		
	SOP	£1	B
START	q0	q1	q2
	q1	q2	q1
	q2		q0

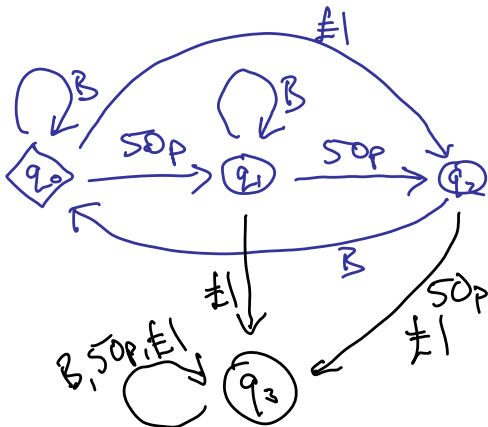


Finite state machines FSM
or Finite state automata DFA

Error states

Can fill in blanks in table by using an error state

	commands		
	SOP	£1	B
q0	q1	q2	q0
q1	q2	q3	q1
q2	q3	q3	q0
q3	q3	q3	q3



Transition function

Now: for every state and every command there is exactly one next state to go to

Mathematically:

- Q = set of states
- Σ = alphabet of commands
- $\delta : Q \times \Sigma \rightarrow Q$ transition function

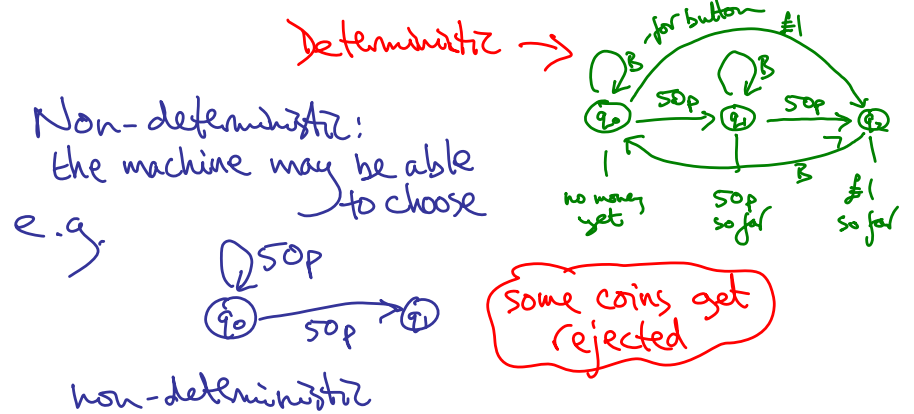
Automaton = machine

But - various different notions

In this module:

automaton = machine with initial & accepting states

Determinism v. non-determinism



Transition ~~function~~ relation - for non-determinism

Now: for every state q and every command x there can be any number of next states to go to

Mathematically:

Q = set of states
 Σ = alphabet of commands
 $\delta \subseteq Q \times \Sigma \times Q$ transition relation

possible next state of $(q, x) \in \delta$

Patterns / regular expressions

color (ϵ/w) r

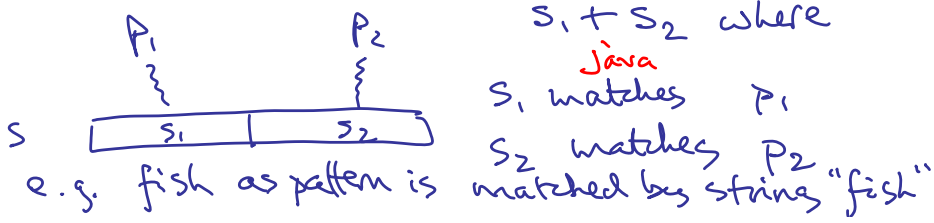
- Empty pattern ϵ
- Letter $a \in \Sigma$
- Concatenation $(p_1 p_2)$ feed sequencing
- Alternative $(p_1 | p_2)$ col(olow) r conditionals
- Kleene Star (p^*) Goo(ok)gle repetition

Strings matching a pattern

Empty pattern ϵ : s matches ϵ if s is empty

Single character a : s matches a if $s = "a"$

Concatenation: $P_1 P_2$ s matches $P_1 P_2$ if can write s as $s_1 + s_2$ where



Strings matching a pattern

pattern = "regular expression"

Alternative $P_1 | P_2$ s matches $P_1 | P_2$ if it matches either P_1 or P_2

e.g. $col(ou)r$ matched by "color" (or $colo(u)r$ and by "colour"

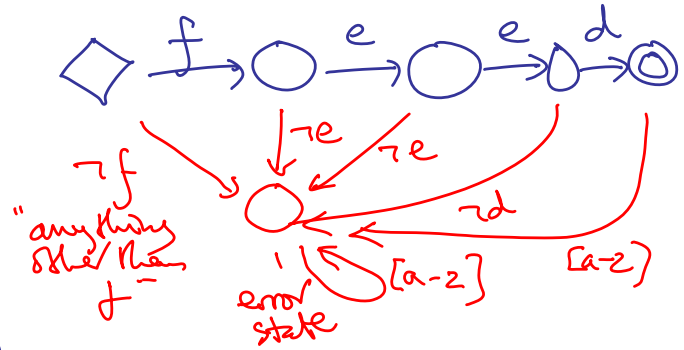
"Kleene star" $(P)^*$ matched by s if $s = ""$ (empty) or s divides as $s_1 + s_2 + \dots + s_n$ where every s_i matches P
 e.g. $\{ (a|b)^* \}$ matched by string with $\{ \epsilon, a, a^2, b, ab, ba, \dots \}$

Using a machine to recognize a pattern

- Use each character as a command
- Build machine with a given "initial" state
- Use string as a sequence of commands
- Machine is such that strings matching the pattern take machine to a special state called an accepting state
- Non-determinism? s matches pattern if it is possible to end in accepting state

Examples

feed

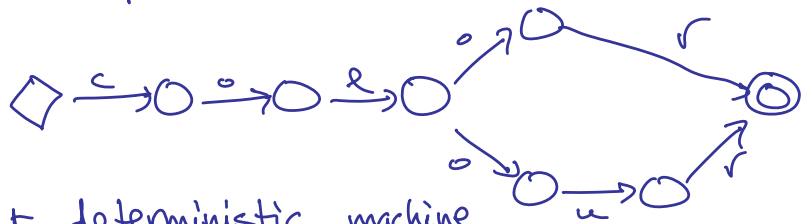


string feed ends in accepting state

feed ends in error state

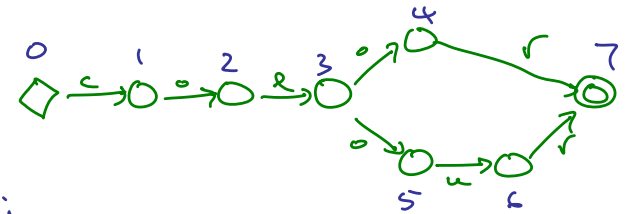
Example

col (o/ou)r



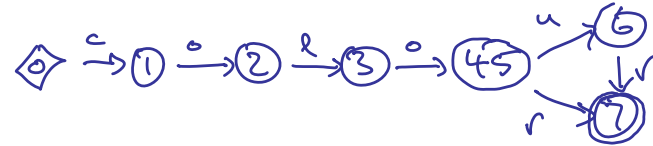
wants deterministic machine

accepting exactly the strings that match the pattern.



Deterministic:

Idea - for non-deterministic transition
- invent a state for set of possibilities

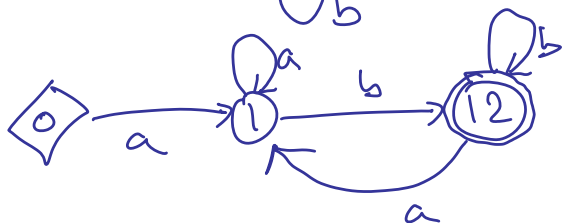


$a (a|b)^* b$

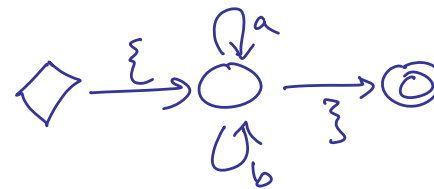
non-deterministic



deterministic

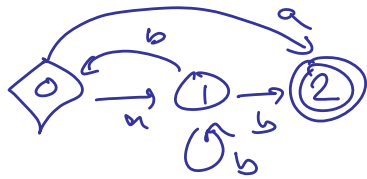


$\{ (a|b)^* \}$ { , then a's & b's, then }

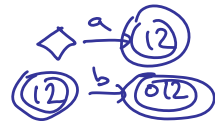


Non-deterministic machine
 \Rightarrow deterministic

e.g.



To build machine: start from \diamond
 bring in other states as needed
 don't need any more



Matches
 $(a^*b^*)^*a^*$ /
 $a^*(b^*a^*)^*b^*$

The following slides

illustrate the general proof (lecture notes)
 that for every pattern there is a deterministic machine

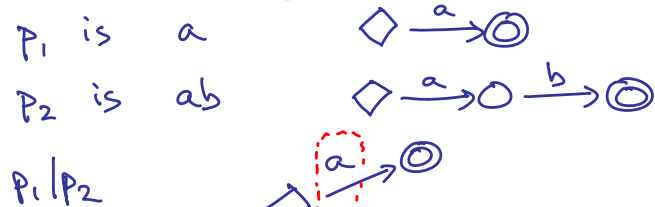
You need to know this fact (emphasized next week)
 You also need to know how - in practice -
 to build the machines.

Naive ideas: ① Alternative



- Problems
- Two accepting states?
 happy with many accepting states
 - but simpler if arrange as just one at first
 - Non-determinacy: will have to fix.

Non-determinacy example



non-determinacy



There's a general way to fix the problem

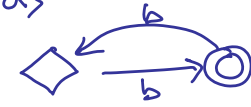
Naive ideas: ② concatenation



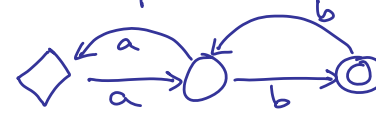
Problem: allows P_1, P_2 to interact.
Accepts more strings than it should

Example

P_1 is $a(aa)^*$ any odd number of a's



$P_1 P_2$: odd no. a's followed by odd no. b's
First attempt



WRONG!

Accepts

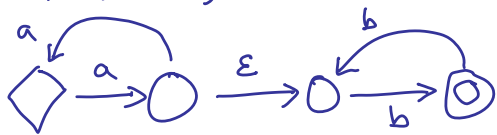
$a(aa|bb)^*b$

e.g. $abbaab$

ϵ -moves

Idea: An ϵ -move can just happen.
It doesn't read from the string.
Use these to decouple different parts
of the automation.

$a(aa)^*b(bb)^*$:



Non-deterministic
again

Strategy

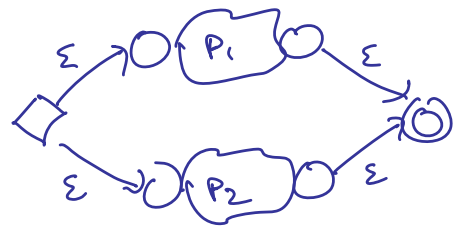
At first: allow non-determinacy
& ϵ -moves

Then: eliminate them

At any stage — remove inaccessible states
— use ingenuity to simplify diagrams

First

P_1 / P_2



$P_1 P_2$



P^*



Eliminating ϵ -moves

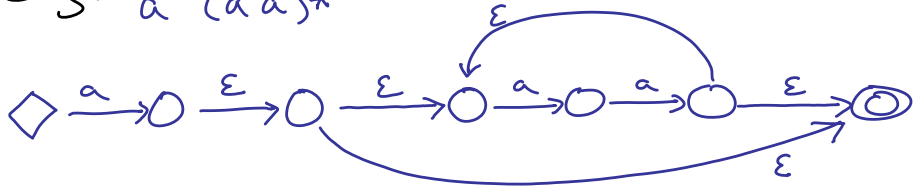
Step 1: change $O \xrightarrow{\epsilon} \odot$ to $\odot \xrightarrow{\epsilon} \odot$

change $O \xrightarrow{\epsilon} O \xrightarrow{a} O$
to $O \xrightarrow{\epsilon} O \xrightarrow{a} O$
with a curved arrow labeled 'a' from the second O to the third O.

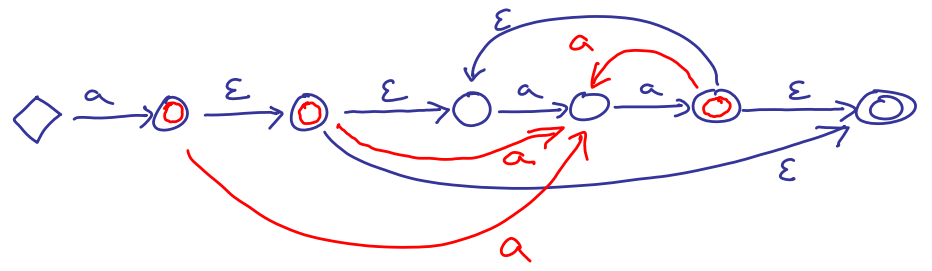
repeat if necessary

Step 2: delete all ϵ -moves

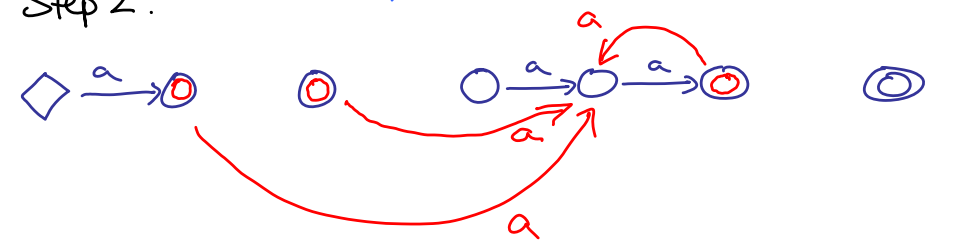
e.g. $a(aa)^*$



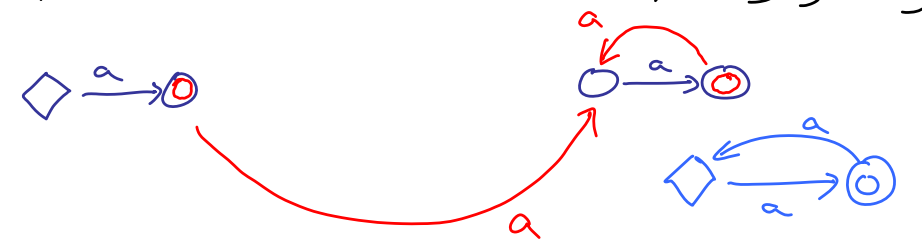
Step 1:



Step 2: $a(aa)^*$

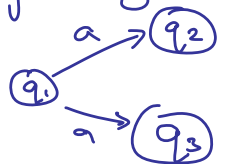



Then remove inaccessible states, apply ingenuity.



Eliminating non-determinacy

Basic idea: use new states that represent sets of original states.

Replace  by 

A new state is accepting if it contains an old accepting state.

Full details in notes.

e.g.

