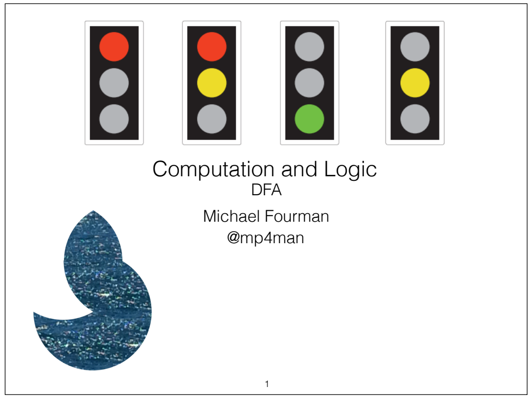


This course provides a first glimpse of the deep connections between computation and logic. We will focus primarily on the simplest non-trivial examples of logic and computation: propositional logic and finite-state machines.

In this lecture we briefly look again at the Wolf Goose and Corn example. We will then look at another example that introduces some ideas that we will explore further in later lectures, and introduce some notation which should become more familiar in due course.

FSM

```
type Sym = Char
type Trans q = (q, Sym, q)
data FSM q = FSM [q] [Sym] [Trans q] [q] [q] deriving Show

-- lift transitions to [q]
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q`elem`ss ]

-- apply transitions for symbol x to move the start states
step :: Eq q => FSM q -> Sym -> FSM q
step (FSM qs as ts ss fs) x = FSM qs as ts (next ts x ss) fs

accepts :: (Eq q) => FSM q -> String -> Bool
accepts (FSM qs as ts ss fs) "" = or[ q`elem`ss | q <- fs ]
accepts fsm (x : xs) = accepts (step fsm x) xs

trace :: Eq q => FSM q -> [Sym] -> [[q]]
trace     (FSM _ _ _ ss _) []     = [ss]
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs
```

A **language** L is a set of strings in some Alphabet Σ

$$L \subseteq \Sigma^*$$

Given an FSM, **M** the language **L(M)** is the set of strings accepted by **M**

A language is **regular** iff it is of the form **L(M)**
i.e. if there is some machine that recognises it

We will see that *some languages are not regular*.

Examples of regular languages:
valid postcodes, strings encoding legal sudoku solutions, binary strings encoding numbers divisible by 17, correct dates in the form Tuesday 13 September 2024 for the entire 20th and 21st centuries

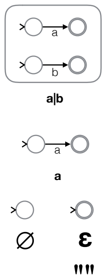
---

language: $L \subseteq \Sigma^*$

is **regular** iff it is of the form **L(M)**

the language {"a"} is regular
the language {"abc"} is regular
the language a* is regular
the language {""} is regular
the language ∅ is regular

if A, B ⊆ Σ * are both regular then so is A∪B — which we also write as A|B



---

simple machines

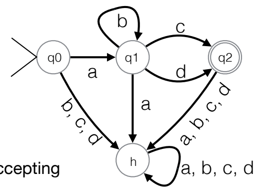
a single start state

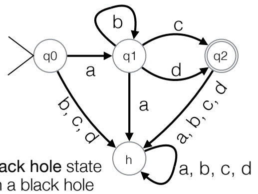
each input sequence leads to a single state

the answer depends only on the this state

for some states, yes **accepting**

for the rest, no

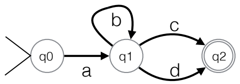




h is a **black hole** state
once in a black hole
we can never escape

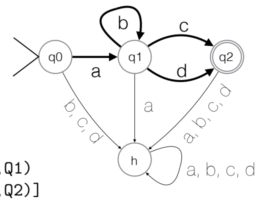
omitting the black hole
gives a simpler diagram

still shows all paths
from start to accepting



DFA
single start state
any number
of accepting states

each (state, input) pair
determines next state



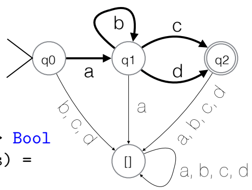

```
ts = [(Q0,a,Q1),(Q1,b,Q1)
     ,(Q1,c,Q2),(Q1,d,Q2)]
```

```
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q`elem`ss ]
next ts a [Q0] = [Q1]
next ts b [Q1] = [Q1]
next ts c [Q1] = [Q2]
next ts d [Q1] = [Q2]
next ts _  _   = [] -- black hole
```

Always, at most one *state is lit*

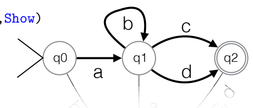
```
type Sym = Char
type Trans q = (q, Sym, q)
data FSM q = FSM [q] [Sym] [Trans q] [q] [q]
             deriving Show
```

DFA



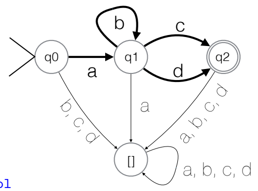

```
isDFA :: Eq q => FSM q -> Bool
isDFA (FSM qs as ts ss fs) =
  (length ss == 1)
  &&
  and[ r == q' | (q, a, q') <- ts, r <- qs
               , (q, a, r)`elem`ts ]
```

```haskell
data EG = Q0|Q1|Q2 deriving (Eq,Show)
[a,b,c,d] = "abcd"
eg = FSM qs as ts ss fs
 where
   qs = [Q0,Q1,Q2]
   as = [a,b,c,d]
   ts = [(Q0,a,Q1),(Q1,b,Q1),(Q1,c,Q2),(Q1,d,Q2)]
   ss = [Q0]
   fs = [Q2]

trace     (FSM _ _ _ ss _) []     = [ss]
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs

> trace eg "abbc"
[[Q0],[Q1],[Q1],[Q1],[Q2]]
> trace eg "abbcd"
[[Q0],[Q1],[Q1],[Q1],[Q2],[]]
```



Always, at most one *state is lit*

DFA

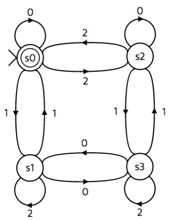



```haskell
isDFA :: Eq q => FSM q -> Bool
isDFA (FSM qs as ts ss fs) =
  (length ss == 1)
  &&
  and[ r == q' | (q, a, q') <- ts, r <- qs, (q, a, r)`elem`ts ]
```

KISS – DFA

Deterministic **F**inite **A**utomaton

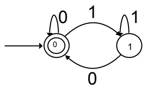
Exactly one start state, and
 from each state, **q**,
  for each symbol, **a**,
   there is
    exactly one transition
    from **q** with label **a**



How can we understand which questions are answered by DFA?

Two examples

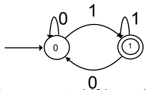
	×2	×2 + 1
	0	1
0	0	1
1	0	1



Even binary numbers

Input sequence is accepted if it ends with a zero.

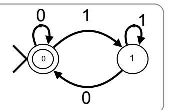
	×2	×2 + 1
	0	1
0	0	1
1	0	1



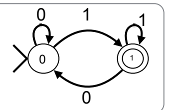
Odd binary numbers

Input sequence is accepted if it ends with a one.

The complement of a regular language is regular

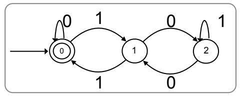


L_0 : even numbers
= 0 mod 2



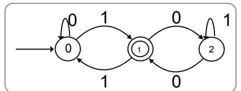
L_1 : odd numbers
= 1 mod 2

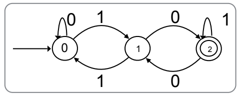
Three examples



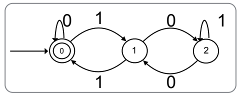
Which binary numbers are accepted?

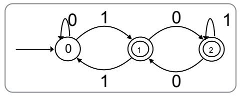
	×2	×2 + 1
mod 3	0	1
0	0	1
1	2	0
2	1	2





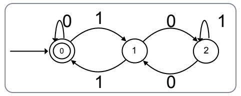
The complement of a regular language is regular



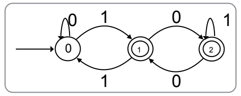


If $A \subseteq \Sigma^*$ is recognised by M then $\overline{A} = \Sigma^* \setminus A$ is recognised by $\overline{M}$ where $\overline{M}$ and M are identical except that the accepting states of $\overline{M}$ are the non-accepting states of M and vice-versa

By three or not by three?

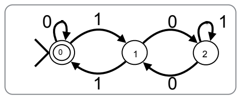


divisible by three



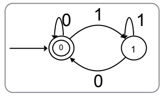
not
divisible by three

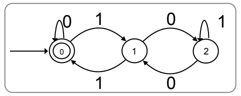
The intersection of two regular languages is regular



$L_0 = 0 \bmod 3$
$L_1 = 1 \bmod 3$
$L_2 = 2 \bmod 3$

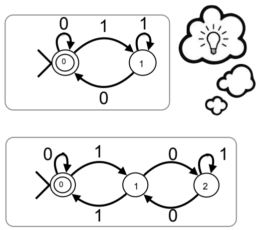
The intersection of two regular languages is regular





divisible by 6
≡
divisible by 2
and
divisible by 3

The intersection of two regular languages is regular

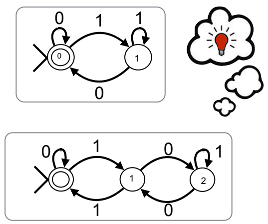


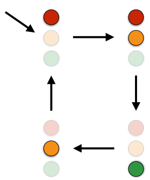
Run both machines in parallel?

Build one machine that simulates two machines running in parallel!

Keep track of the state of each machine.

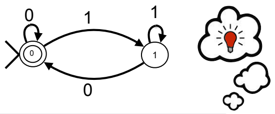
The intersection of two regular languages is regular

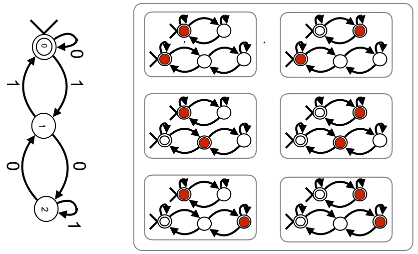




intersection of languages

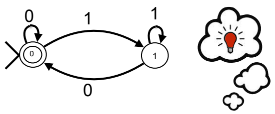
run the two machines in parallel
when a string is in both languages,
both are in an accepting state

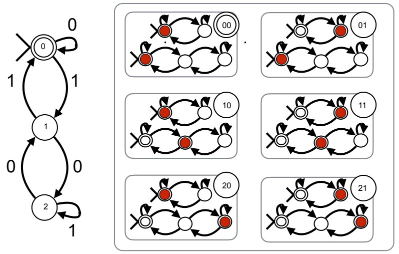




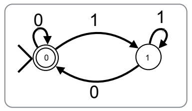
intersection of languages

run the two machines in parallel
when a string is in both languages,
both are in an accepting state

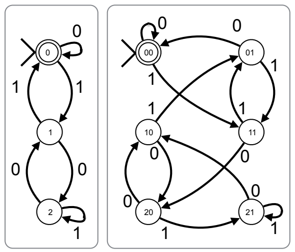




intersection of two regular languages is regular



run two machines in synchrony



The languages $A \subseteq \Sigma^*$ recognised by a DFA form a Boolean Algebra

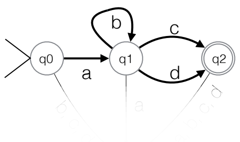
- Since they are closed under intersection and complement.

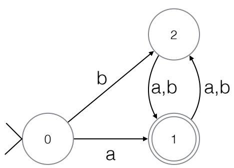
Given a string we can check whether the machine accepts it

How can we describe the strings this machine accepts?

```
> accepts eg "abc"
True
> accepts eg "abbd"
True
> accepts eg "abcd"
False
```

`ab*(c|d)`



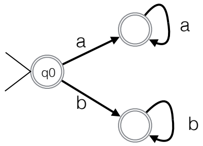


`(a|(b(a|b))((a|b)(a|b))*`

a*|b*

---

a*|b*



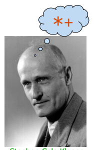
Plus a black hole state

---

regular expressions
patterns that match strings

- any character is a regexp
  - matches itself
- if R and S are regexps, so is RS
  - matches
    a match for R followed by a match for S
- if R and S are regexps, so is R|S
  - matches
    any match for R or S (or both)
- if R is a regexp, so is R*
  - matches
    any sequence of 0 or more matches for R
- The algebra of regular expressions also includes elements ∅ and ε
  - ∅ matches nothing;
  - ε = ∅* matches the empty string

Kleene *, +



Stephen Cole Kleene

1909-1994

The union of two regular languages is a regular language

The empty language is a regular language

The all-inclusive language is a regular language

The complement of a regular language is a regular language?

Any Boolean combination of regular languages is a regular language