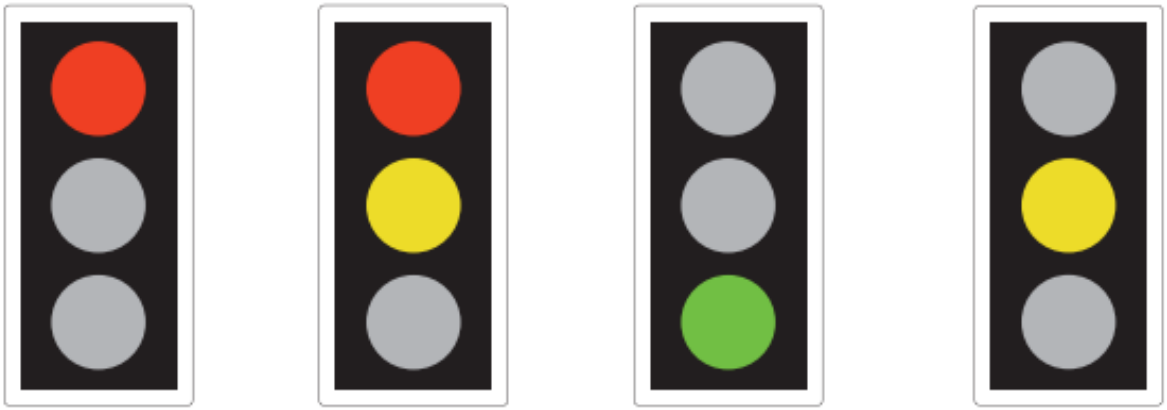


Computation and Logic

DFA

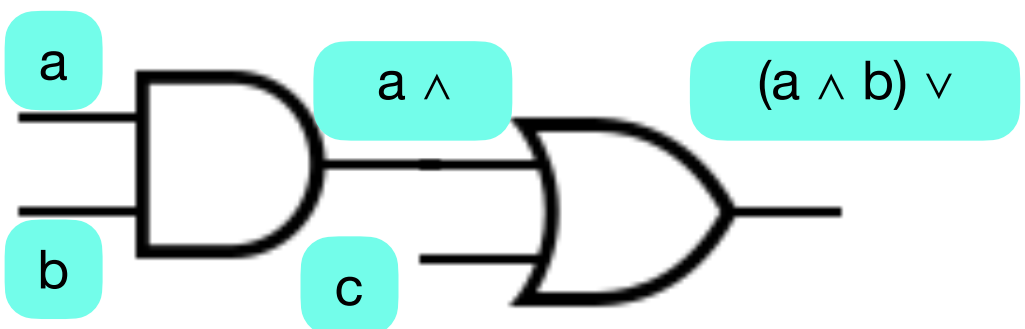
Michael Fourman
@mp4man

Survey Monkey Link

If we start from an expression then
we can draw an equivalent circuit with:

$R = (X \wedge Y) \vee Z$

a wire for each subexpression,
a logic gate for each operator,
and an input for each variable.

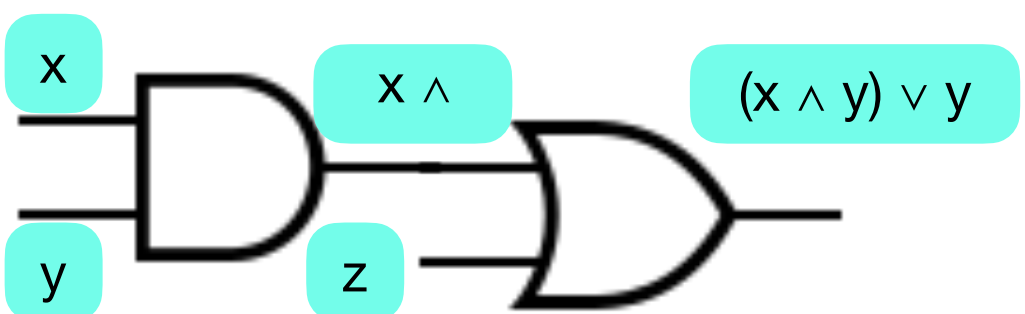


V a ∧ = V a ∧ V b

If we start from an expression then
we can draw an equivalent circuit with:

$R = (X \wedge Y) \vee Z$

a wire for each subexpression,
a logic gate for each operator,
and an input for each variable.

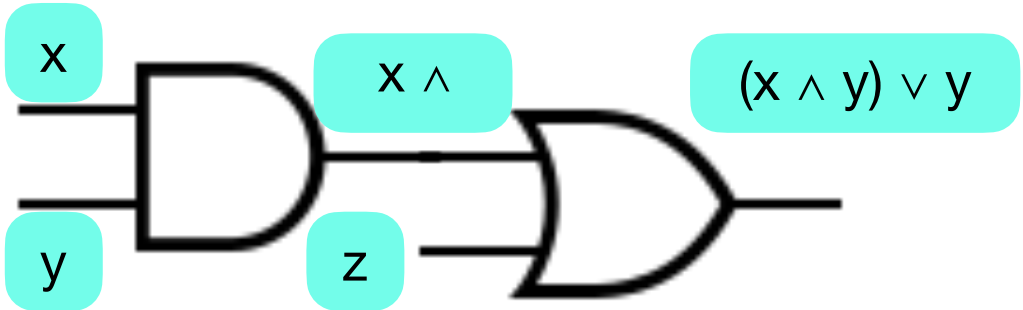


V x ∧ **=** **V** x ∧ **V** y

V (x ∧ y) ∨ z **=** **V** x ∧ ∨ **V** z

Relationships between the values on the wires

$$R = (X \wedge Y) \vee Z$$



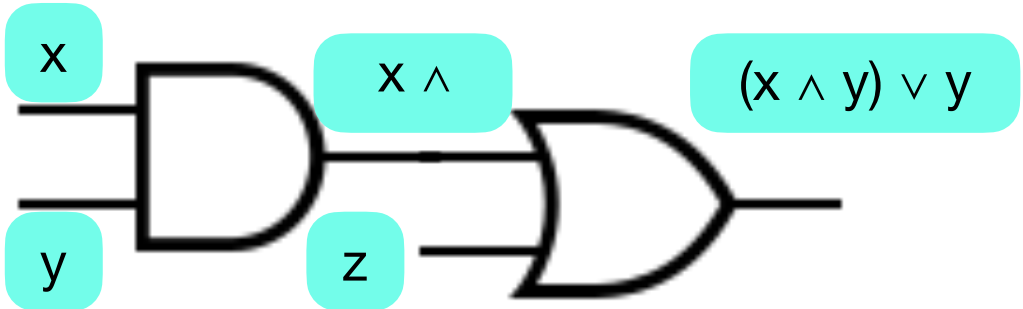
Relationships between the values on the wires

$$\mathbf{V}(x \wedge) = \mathbf{V}(x) \wedge \mathbf{V}(y)$$
$$\mathbf{V}((x \wedge y) \vee z) = \mathbf{V}(x \wedge) \vee \mathbf{V}(z)$$

The following expressions must be true

$$\mathbf{V}(x \wedge) \leftrightarrow \mathbf{V}(x) \wedge \mathbf{V}(y)$$
$$\mathbf{V}((x \wedge y) \vee z) \leftrightarrow \mathbf{V}(x \wedge) \vee \mathbf{V}(z)$$

$$R = (X \wedge Y) \vee Z$$



The following expression must be true

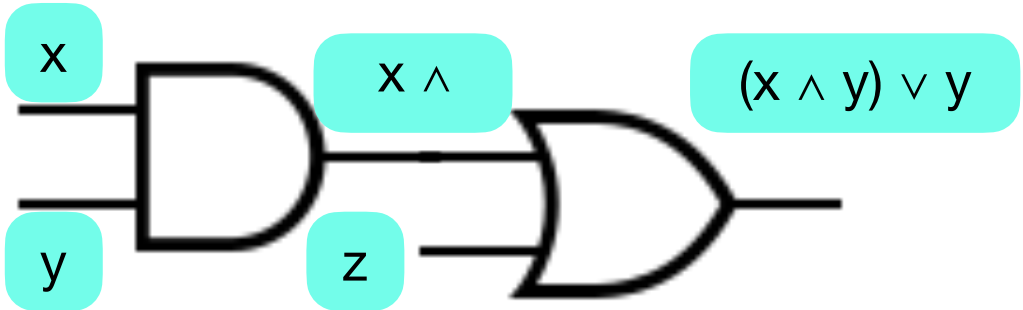
r **a** **b**

V (x ∧ y) ∨ z ↔ **V** x ∧ ∨ **V** z

```
*CL7> [x,y,z] = "xyz"
*CL7> r@(a :|: b) = (V x :&: V y) :|: V z
*CL7> r
V 'x' :&: V 'y' :|: V 'z'
*CL7> a
V 'x' :&: V 'y'
*CL7> b
V 'z'
*CL7> V r :<->: V a :|: V b
V (V 'x' :&: V 'y' :|: V 'z') :<->: V (V 'x' :&: V 'y') :|: V (V 'z')
```

r a b

$$R = (X \wedge Y) \vee Z$$



The following expression must be true

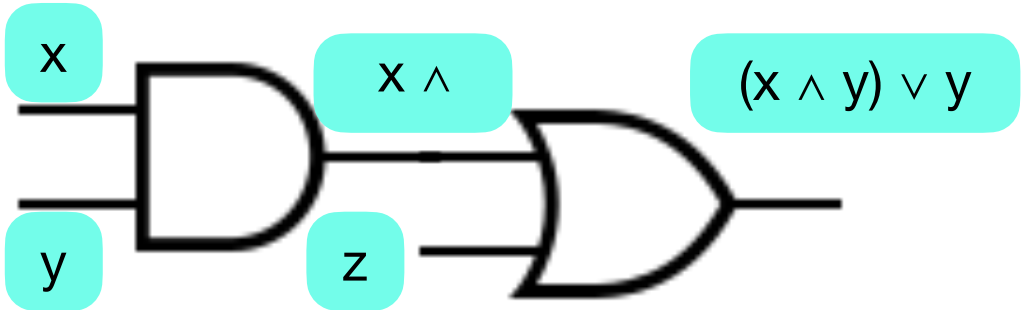
r **a** **b**

V (x ∧ y) ∨ z ↔ **V** x ∧ ∨ **V** z

```
*CL7> [r,a,b] = "rab"
*CL7> wffToForm (V r :<->: V a :|: V b)
And [Or [N a,P r],Or [N b,P r],Or [N r,P a,P b]]

tt r@(a :|: b)    = [ Or[N r, P a, P b]
                    , Or[P r, N a], Or[P r, N b]]
                    ++ tt a ++ tt b
```

$$R = (X \wedge Y) \vee Z$$



The following expression must be true

$$\mathbf{V}\ (x \wedge y) \vee z \leftrightarrow \mathbf{V}\ x \wedge \ \vee\ \mathbf{V}\ z$$

```
tt :: Ord a => Wff a -> Form (Wff a)
tt r@(Not a)   = wffToForm (V r :<->: Not (V a))
                   <&&> tt a
tt r@(a :&: b) = wffToForm (V r :<->: V a :&: V b)
                   <&&> tt a <&&> tt b
tt r@(a :|: b) = wffToForm (V r :<->: V a :|: V b)
                   <&&> tt a <&&> tt b
```

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*.

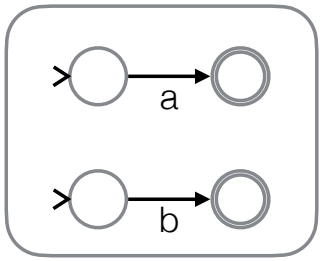
Examples of regular languages:

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

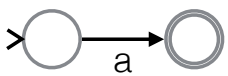
language: L ⊆ Σ *

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



a|b



a

∅ **ε** **""**

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

operations on languages:

`for,` $A, B \subseteq \Sigma^*$ we have the Boolean operations:

`alternation` $A \mid B = A \cup B$ `intersection` $A \cap B$

`difference` $\neg A = \Sigma^* \setminus A$ `nothing` $\varnothing$ `everything` $\top = \Sigma^*$

and some more, `concatenation` AB, and, `iteration` A^* .

`concatenation` is easy: $AB = \{a\texttt{++}b \mid a \in A, b \in B\}$

A^* is defined by two rules: $\dfrac{}{\texttt{""} \in A^*}$ $\dfrac{s \in A^* \quad a \in A}{s\texttt{++}a \in A^*}$

or, equivalently, by: $\dfrac{}{\texttt{""} \in A^*}$ $\dfrac{s \in A^* \quad a \in A}{a\texttt{++}s \in A^*}$

because, $\texttt{""++}a_1\texttt{++}a_2 \ldots \texttt{++}a_n = a_1\texttt{++}a_2 \ldots \texttt{++}a_n\texttt{++""}$.

simple machines

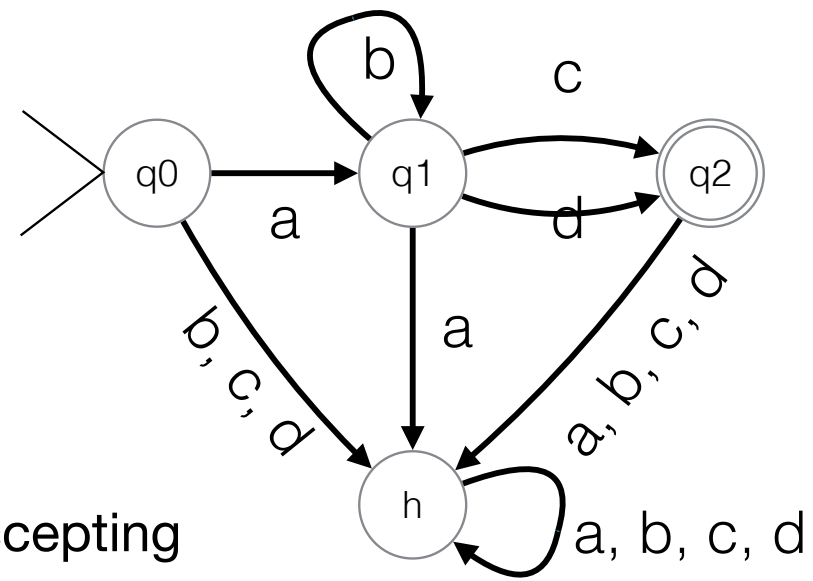
a single start state

each input sequence
leads to a single state

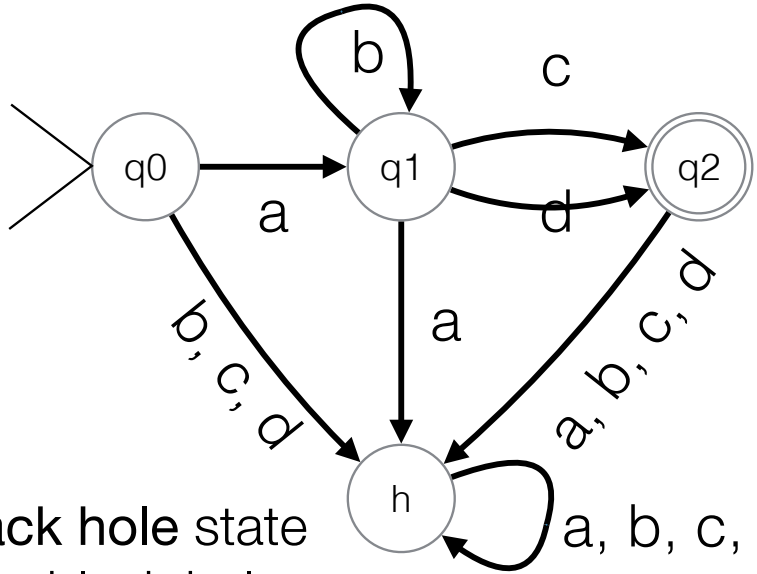
the answer depends
only on the this state

for some states, yes

for the rest, no



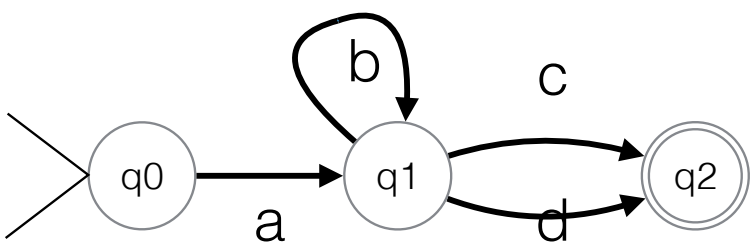
accepting



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

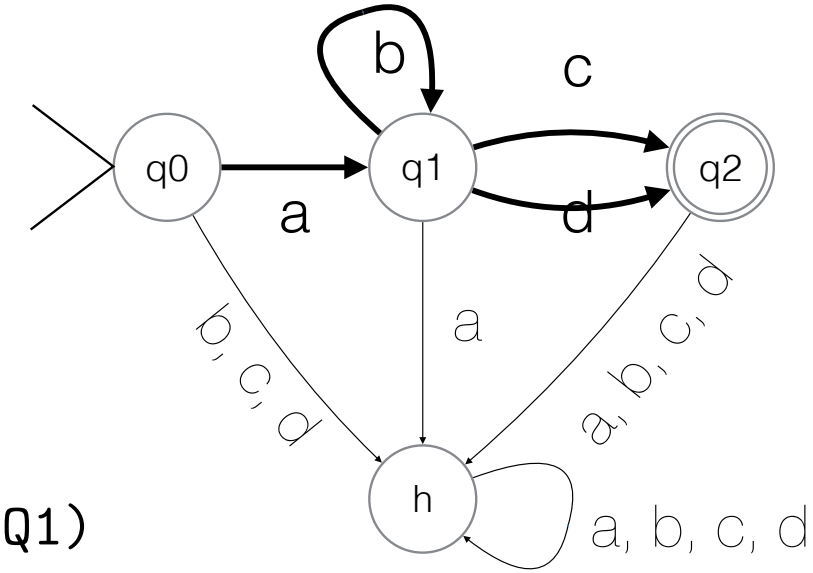
omitting the black hole
gives a simpler diagram

still shows all paths
from start to accepting



DFA
single start state
any number
of accepting states

each (state, input) pair
determines next state



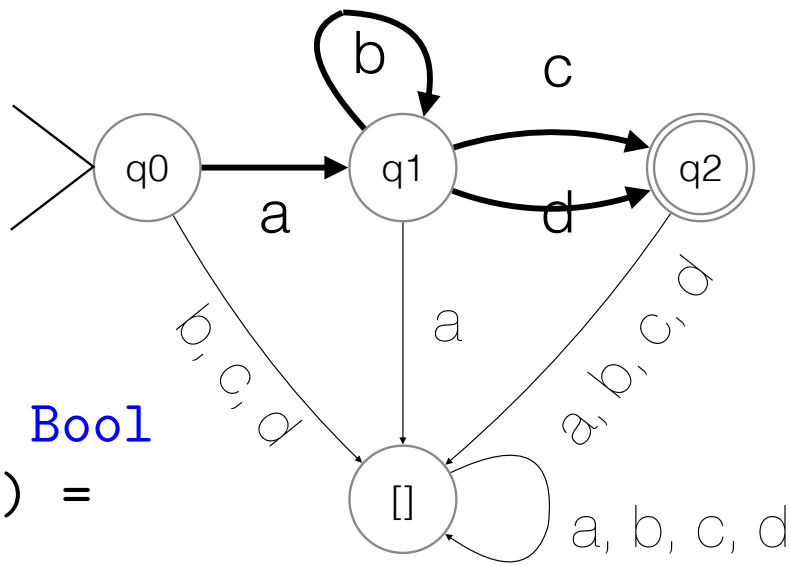

```
ts = [(Q0,a,Q1),(Q1,b,Q1)
     ,(Q1,c,Q2),(Q1,d,Q2)]
```

```
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*

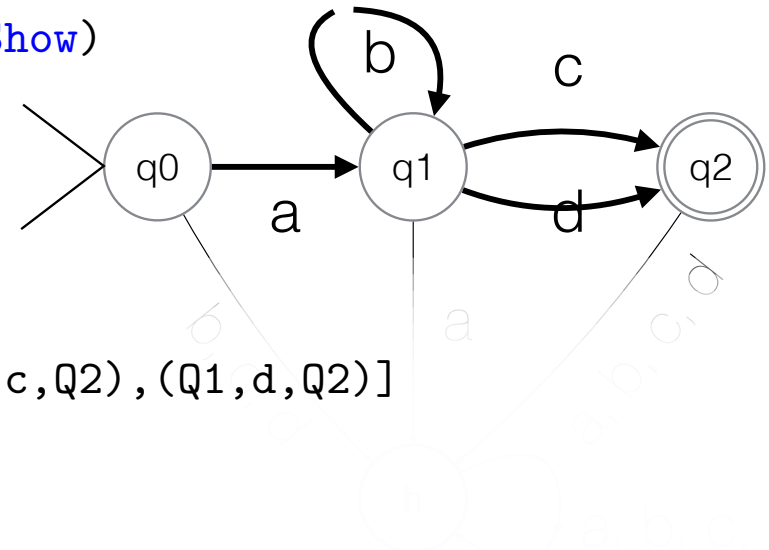
```
type Sym = Char
type Trans q = (q, Sym, q)
data FSM q = FSM [q] [Sym] [Trans q] [q] [q]
             deriving Show
```

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 ]
```

```
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]
```

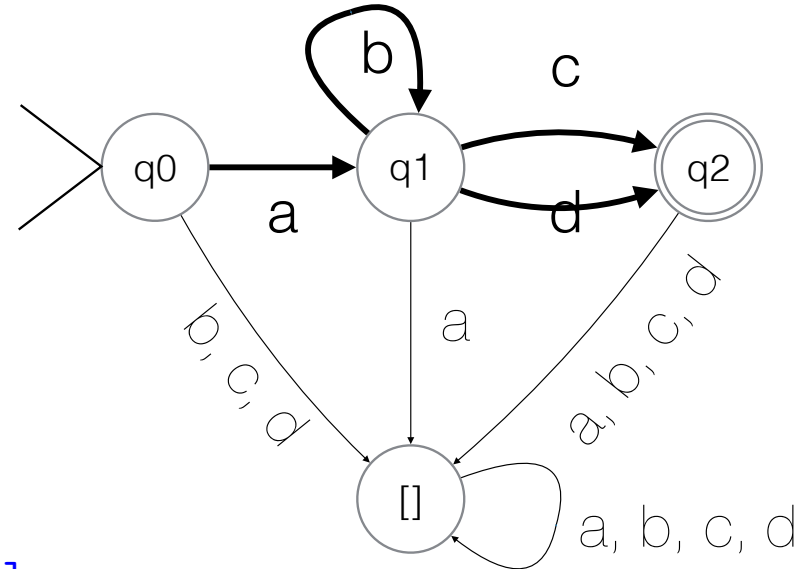



```
trace     (FSM _ _ _ ss _) []     = [ss]
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs

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

Always, at most one *state is lit*

DFA

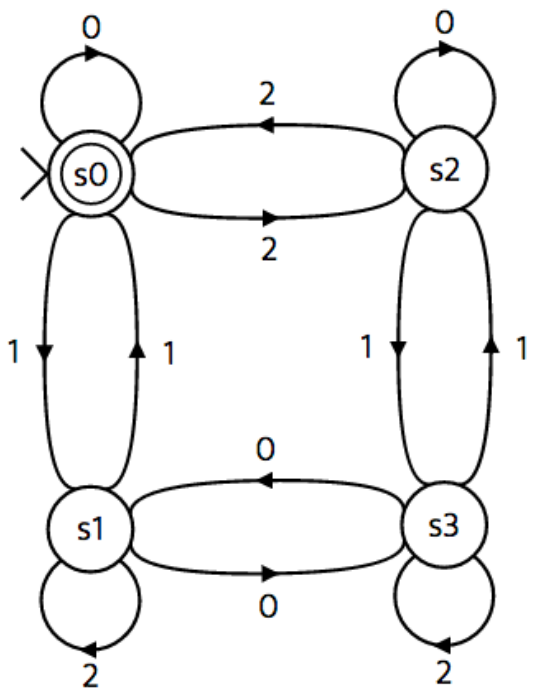



```
isDFA :: Eq q => FSM q -> Bool
isDFA (FSM qs as ts ss fs) =
  (length ss == 1)
  &&
  and[ r == q' | (q, a, q') <- ts, r <- qs, (q, a, r)`elem`ts ]
```

KISS – DFA

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

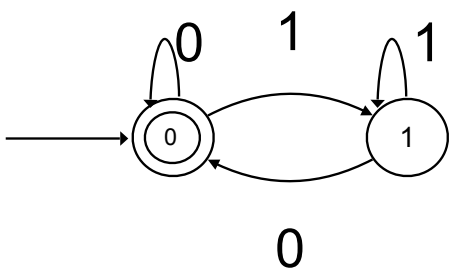
Exactly one start state, and
from each state, **q**,
for each symbol, **a**,
there is
exactly one transition
from **q** with label **a**



How can we understand which questions are answered by DFA?

Two examples

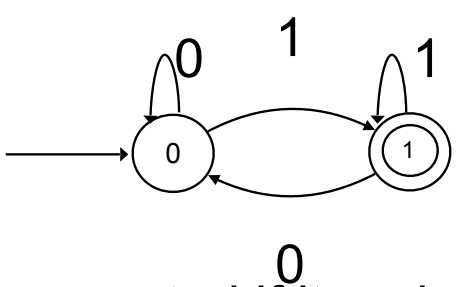
	×2	×2 + 1
	0	1
0	0	1
1	0	1



Even binary numbers

Input sequence is accepted if it ends with a zero.

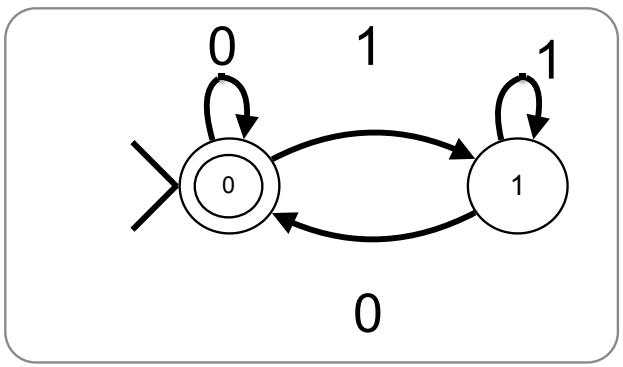
	×2	×2 + 1
	0	1
0	0	1
1	0	1



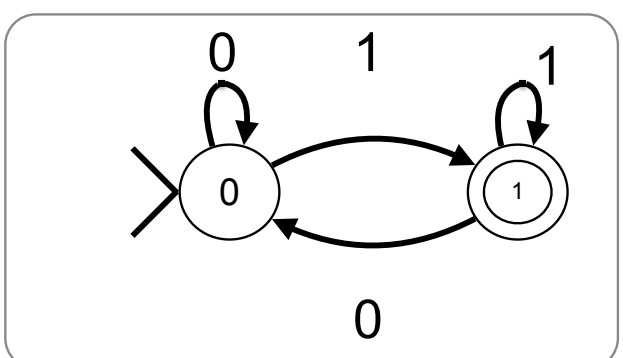
Odd binary numbers

Input sequence is accepted if it ends with a one.

The complement of a DFA regular language is DFA regular

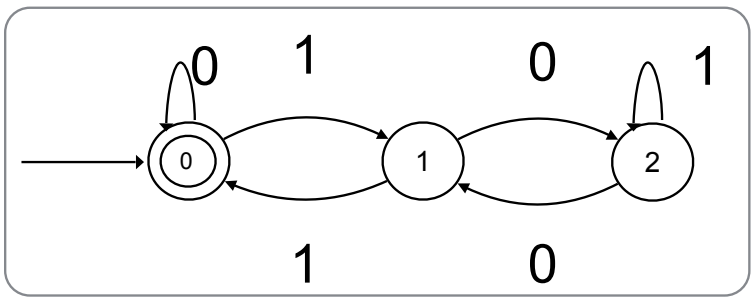


L_0 : even numbers
= 0 mod 2



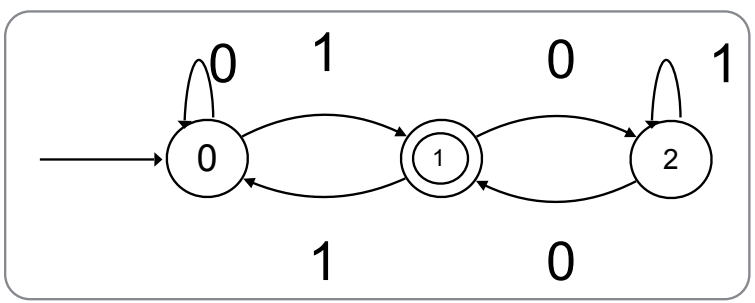
L_1 : odd numbers
= 1 mod 2

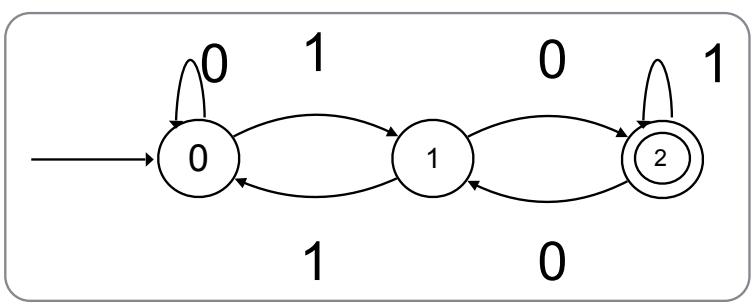
Three examples



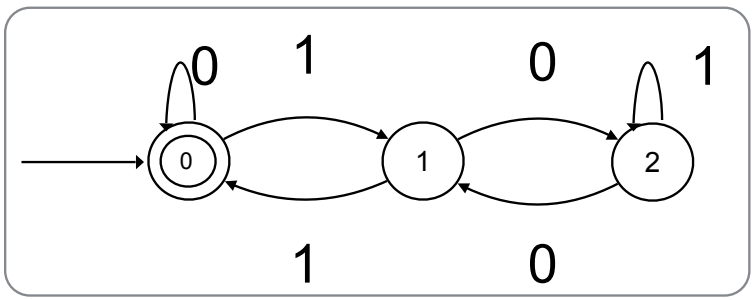
Which binary numbers are accepted?

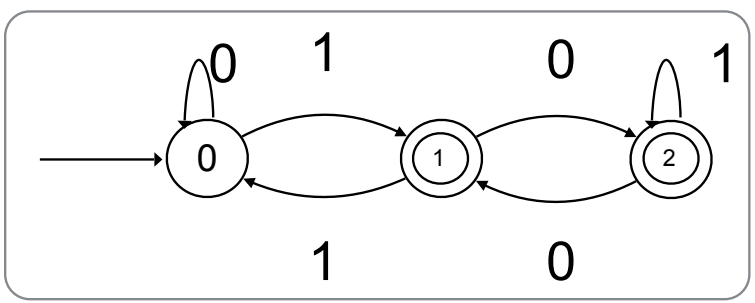
mod 3	×2	×2 + 1
	0	1
0	0	1
1	2	0
2	1	2





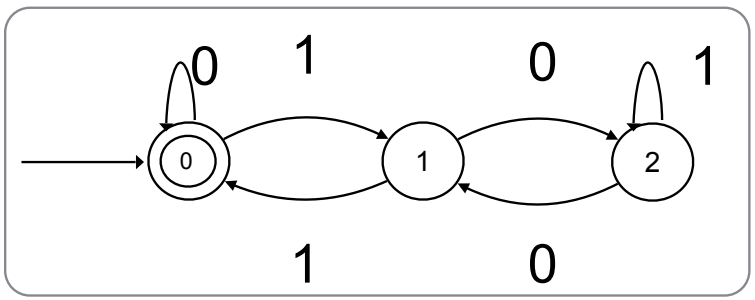
The complement of a DFA regular language is DFA regular



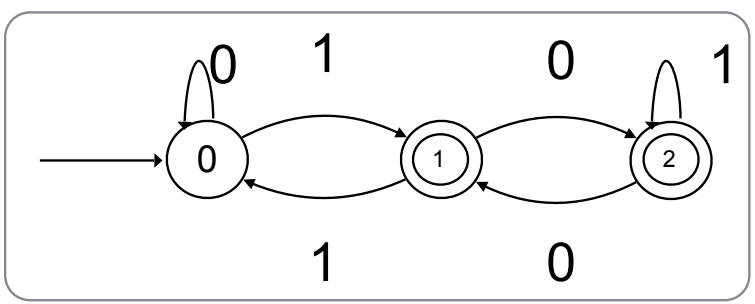


If $A \subseteq \Sigma^*$ is recognised by M
then $\overline{A} = \Sigma^* \setminus A$
is recognised by
$\overline{M}$
where $\overline{M}$ and **M** are identical except that the accepting states of $\overline{M}$ are the non-accepting states of **M** and vice-versa

By three or not by three?

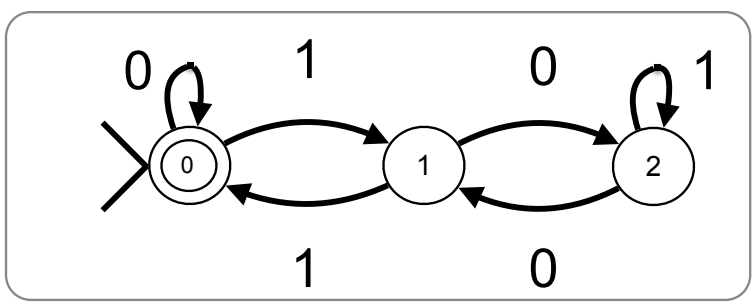


divisible by three



not
divisible by three

The intersection of two DFA regular languages is DFA regular

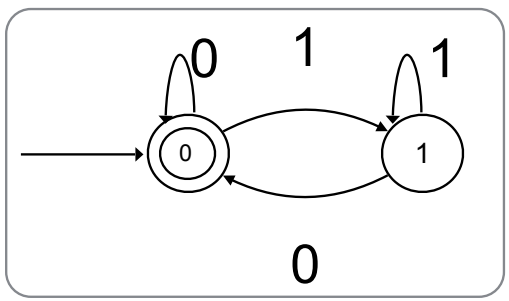


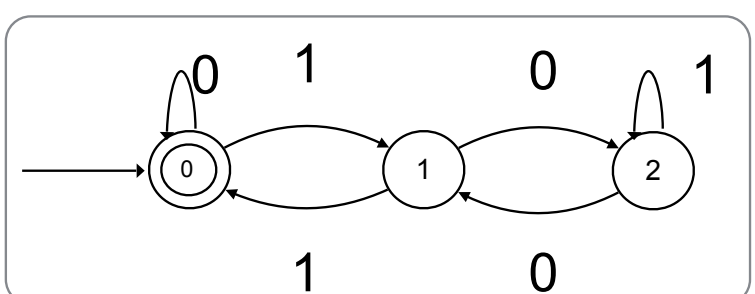
$L_0 = 0 \bmod 3$

$L_1 = 1 \bmod 3$

$L_2 = 2 \bmod 3$

The intersection of two DFA regular languages is DFA regular





divisible by 6

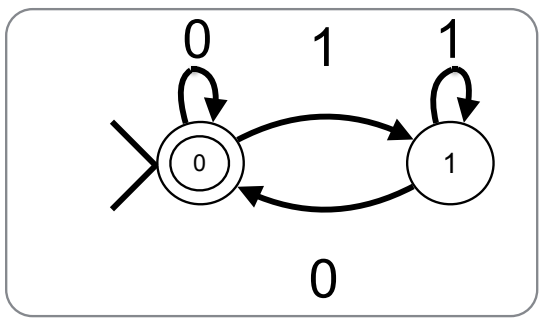
≡

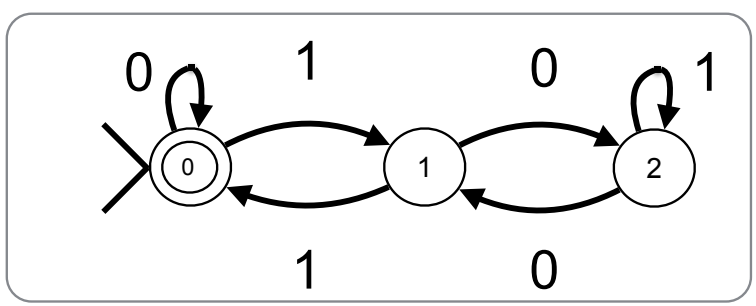
divisible by 2

and

divisible by 3

The intersection of two DFA-regular languages is DFA-regular



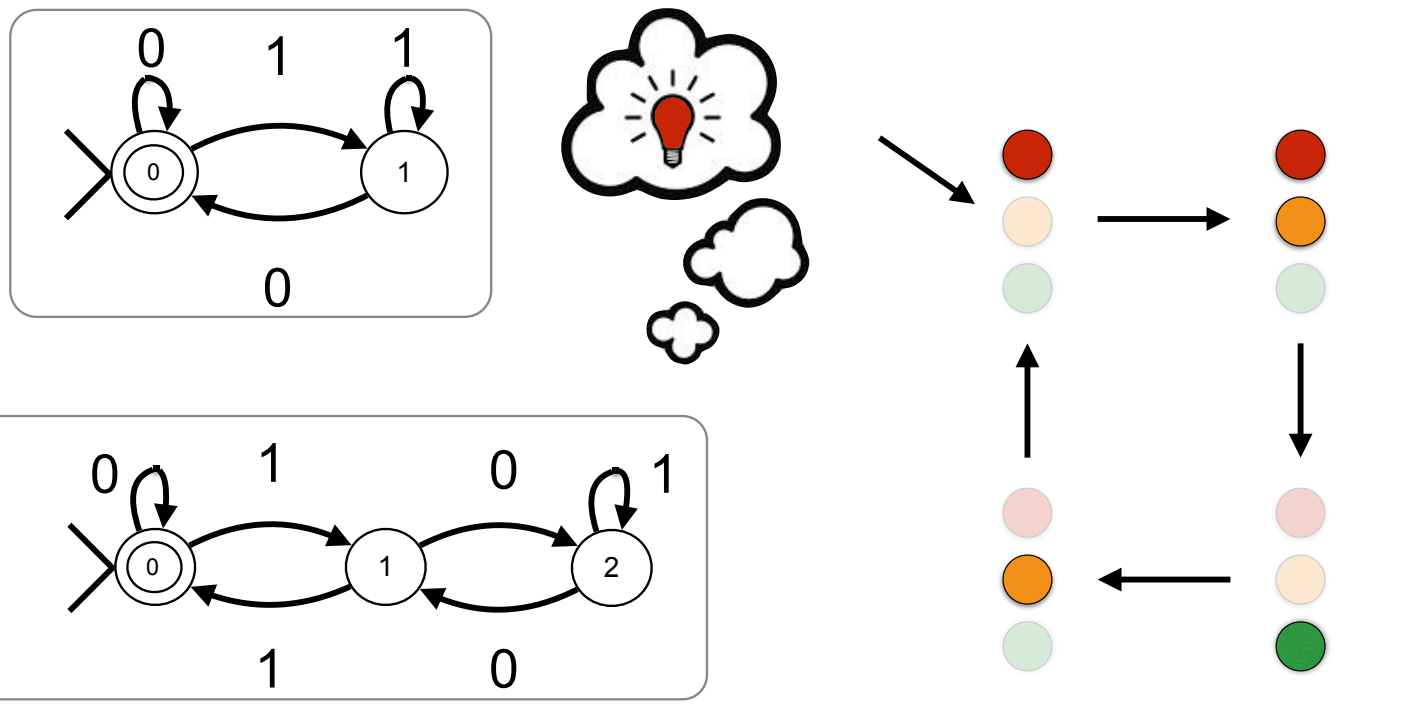


Run both machines in parallel?

Build one machine that simulates two machines running in parallel!

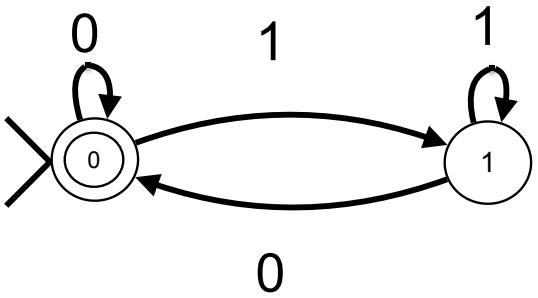
Keep track of the state of each machine.

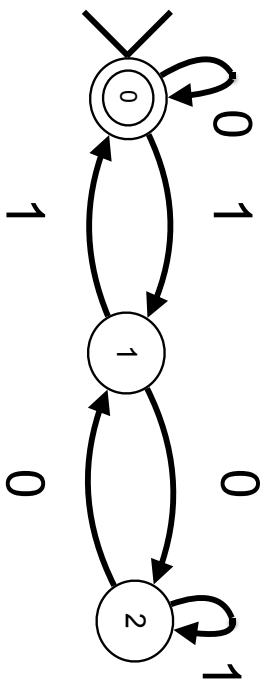
The intersection of two DFA-regular languages is DFA-regular

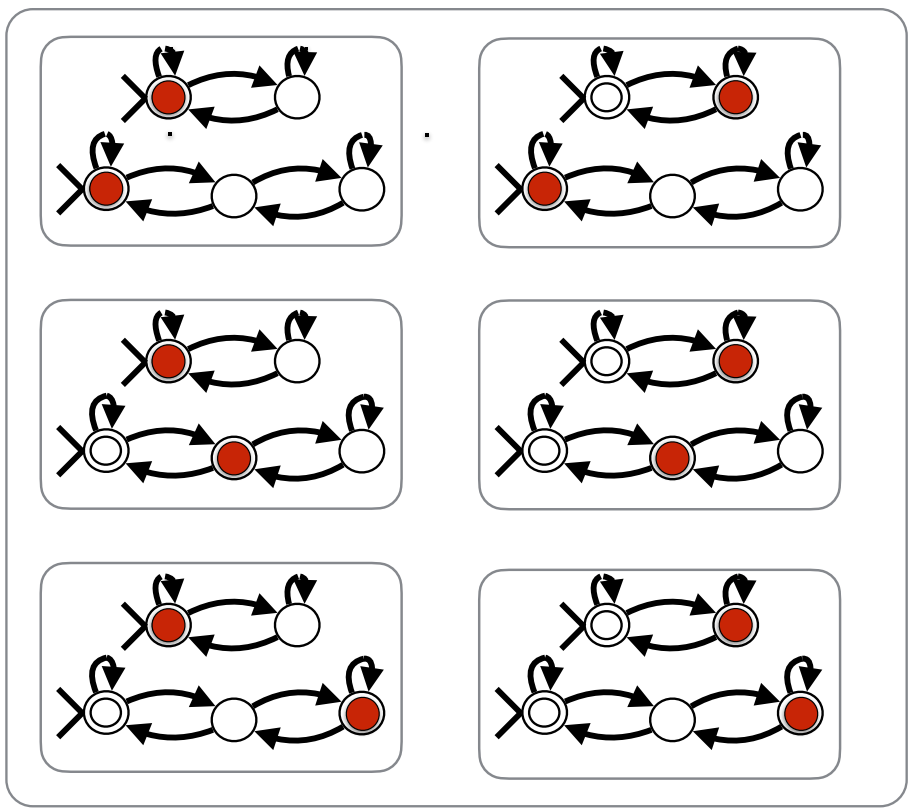


intersection of languages

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

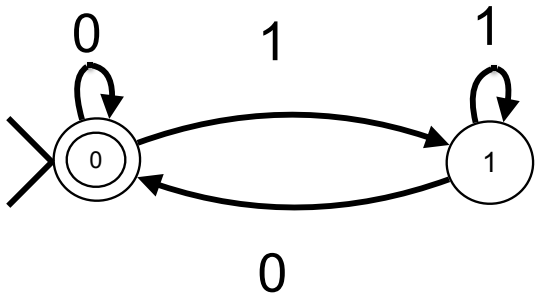


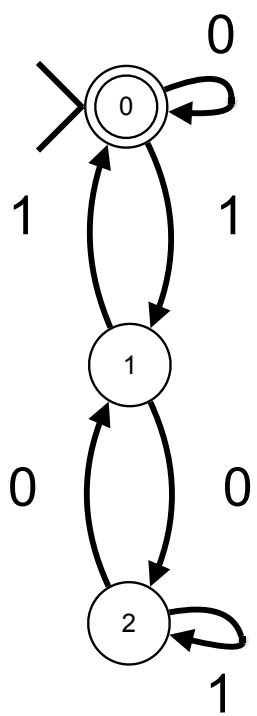


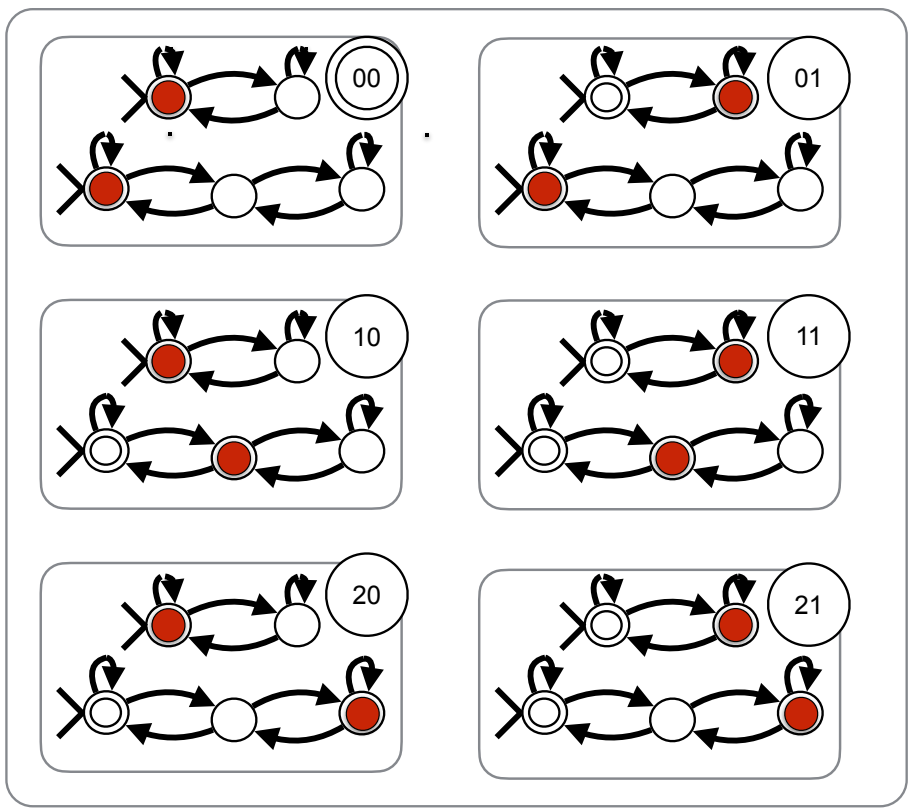


intersection of languages

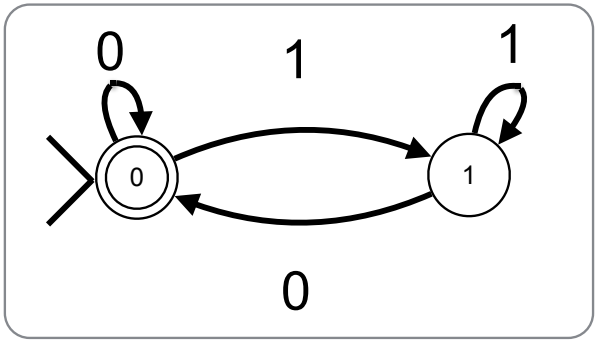
run the two machines in parallel
when a string is in both languages,
both are in an accepting state



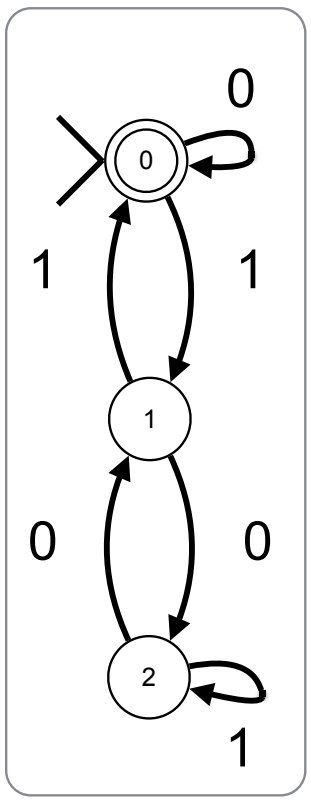


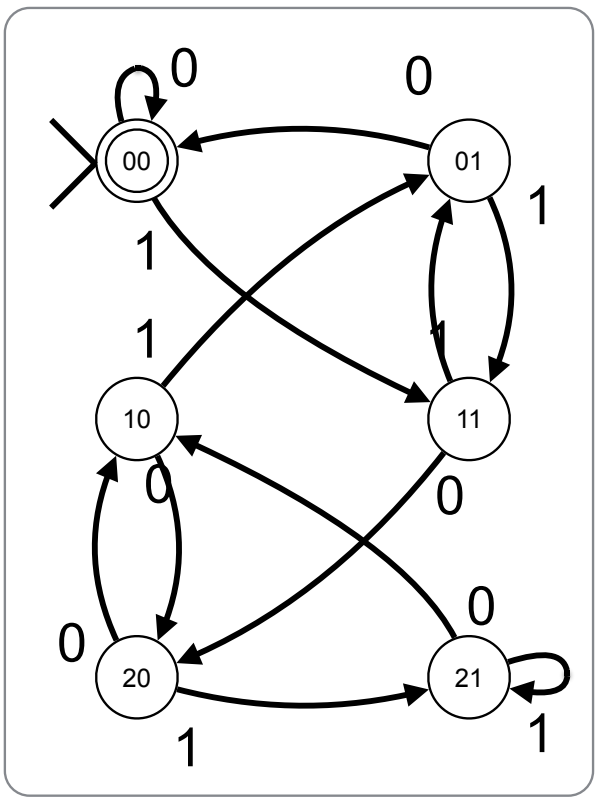


intersection of two regular languages is regular



run two machines in synchrony





The DFA-regular languages $A \subseteq \Sigma^*$ form a Boolean Algebra

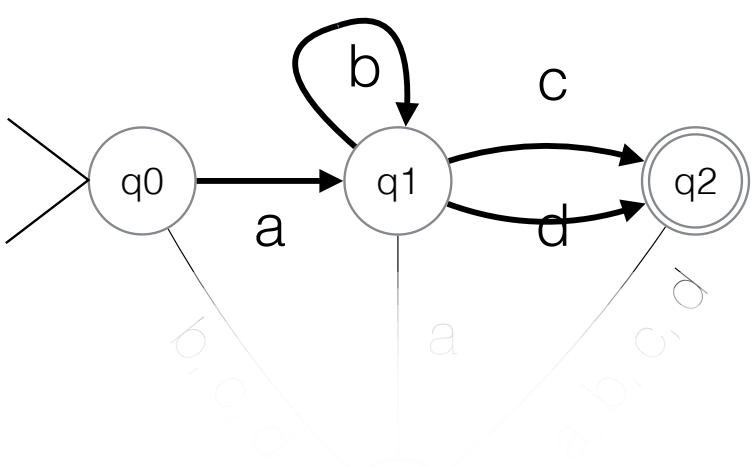
- Since they are closed under intersection and complement.

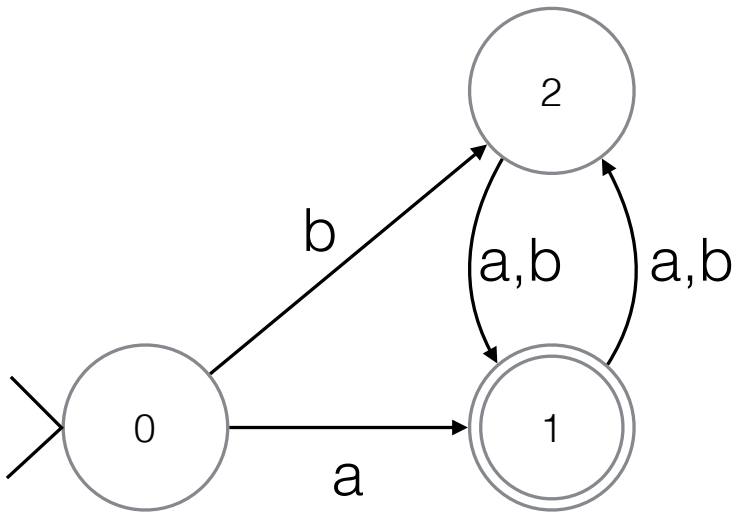
Given a string we can check whether the machine accepts it

How can we describe the strings this machine accepts?

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

ab*(c|d)

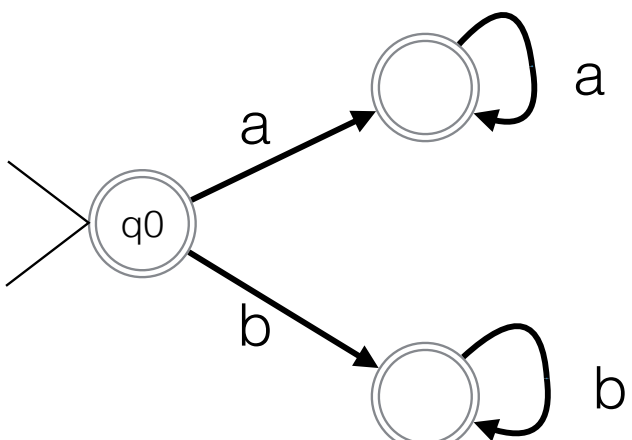




(a|(b(a|b))((a|b)(a|b))*

a*|b*

a*|b*



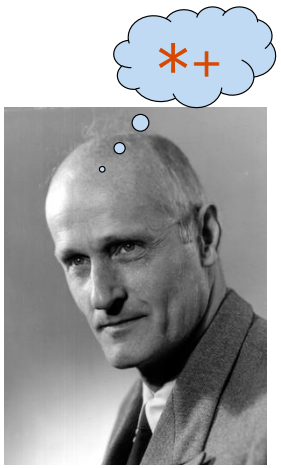
Plus a black hole state

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 DFA regular language is a regular language?

Any Boolean combination of DFA regular languages is a DFA regular language

```
dfa :: Ord q => FSM q -> FSM [q]
dfa (FSM qs as ts ss fs) =
  let superss = reach (next ts) as ss
      superts  = [ (qq, a, next ts a qq) | qq <- superss, a <- as ]
  in
   FSM superss as superts [ss]
   [ qq | qq <- superss, or[ q`elem`fs | q <- qq ]]
```

```
reach :: Ord q => (Sym -> [q] -> [q])
      -> [Sym] -> [q] -> [[q]]
reach step as ss =
  let add qss qs = if qs`elem`qss then qss
        else foldl add (qs : qss)
             [ canonical $ step s qs | s <- as ]
  in add [] (canonical ss)

dfa :: Ord q => FSM q -> FSM [q]
dfa (FSM qs as ts ss fs) =
  let superqs = reach (next ts) as ss
      superts  = [ (qq, a, next ts a qq)
                 | qq <- superqs, a <- as ]
  in FSM superqs as superts [ss]
     [ qq | qq <- superqs, or[ q`elem`fs | q <- qq ]]
```