Overview and Motivation

Course objectives 1

In this module we explore a number of fundamental questions.

- *How can we model a computer, mathematically?*

  Answer should be independent of particular hardware. Focus on the essence. As something that can carry out various instructions.

- *What is an algorithm?*

  Mathematically define the set of instructions mentioned above.

Course objectives 2

We will introduce a few different models of computation that were proposed (in some cases before the first physical computers have been built).

They have names like *finite automata, Turing machines, pushdown automata.* This brings us to other questions:

- *How do we know a given model of computation is the "right" one?*
- *How should we compare different models of computation?*

Comparing models of computation

- We can compare models of computation according to how *powerful* they are.

- Intuitively one model is more powerful than another if it can solve more problems.

- In fact in the theory of computation, the most basic “problems” we consider always boils down to a yes/no question of the form: *does this string of symbols belong to a given set?*

- We call the given set a *language,* and say the model of computation is able to *recognise* the language if it can correctly answer every yes/no question concerning that language.

- Then a model of computation is more powerful if it can recognise more languages.

Course objectives 3

All this talk of measuring the power of a model of computation leads on to another fundamental question regarding the *limits* of computation:

- *Are there languages that no computer can recognise?*

We will see the answer, surprisingly, is yes!

Being able to establish this is one of the benefits of giving a rigorous, mathematical definition of a computer.

Value of the module

In 20 years, computers and programming will be vastly different. But this material will be the same - and will still be useful.

Provides insight into fundamental questions:

- Defines the questions
- Answers some
- Many are still *open*!
- Close connections with logic, algorithms, linguistics, others.

More value of the module

- Provides advanced problem-solving tools:
    - Springboard for more advanced courses, e.g., *Artificial Intelligence, Combinatorial Optimisation*
    - Research
    - Applications

- Practice with mathematics and proofs

- Trace the footsteps of some of the great computer pioneers.

Alan Turing

Other questions we'll look at

- How can we reason about the correctness of a given computer program and prove that it's correct?
- How could we encode *logical reasoning* in a computer?
- How *hard* is it to compute the answers to a given problem?

Set theory, developed as a foundation for all of mathematics, provides a very useful formal framework in which to express the answers to such questions.

Set theory

- union: $S \cup T$
- intersection: $S \cap T$
- empty set: $\varnothing$
- set difference: $S \setminus T$ or $S - T$
- complement: $\overline{S}$

- subset: $S \subseteq T$
- element of: $x \in S$

- distributivity: $S \cup (T \cap V) = (S \cup T) \cap (S \cup V)$
  $S \cap (T \cup V) = (S \cap T) \cup (S \cap V)$
- comprehension: $\{x \in S \mid P(x)\}$ or $\{x \in S : P(x)\}$
  the set of elements of S satisfying property P

Application: Computer languages

- Basis for tools and programming techniques
  - Lexical analysis
  - Parsing
  - Program analysis

- Many interesting problems in programming language implementations are hard or impossible to solve in general. E.g., equivalence of grammars.

Application: Formal verification

- Formal verification attempts to *prove* system designs (e.g., programs) correct, or to find bugs.
- Methods are generally from logic and automata theory. Many of the constructions in this course are used in practical tools.
- Again, many problems in this area are hard or impossible to solve. (See *complexity theory* to study why.)

Basic Concepts: Languages

Basic concepts

- To begin with, we will build up to our formal definition of a language, via the basic concepts of *alphabet* and *string.*
- Along the way we'll introduce the idea of defining things, and proving properties *by induction*.

Alphabets

DEFINITION
An alphabet is a non-empty finite set. The members of the alphabet are called symbols.

Examples

- Binary alphabet {0,1}
- ASCII character set - the first 128 numbers, many of which are printed as special characters.

The capital Greek sigma (Σ) is often used to represent an alphabet.

Strings

Informally: A string is a finite sequence of symbols from some alphabet.

Examples

- ε - the empty string (same for every alphabet).

  (Leaving a blank space for the empty string is confusing, so we use the Greek letter "epsilon").

  ε is not a symbol in the alphabet! It is the string with no symbols; the string of zero length.

Strings, continued

Examples (contd)

- 000, 01101 are strings over the binary alphabet.
- "String" is a string over the ASCII character set, or the English alphabet.

We can formally define what it means for something to be a "string over some alphabet" in different ways. We give the following definition so as to illustrate the important idea of defining things via *induction*.

Strings over an alphabet

DEFINITION (strings over alphabet ∑)

Base: ε is a string over ∑.
Induction: If *x* is a string over ∑ and *a* is a symbol from ∑, then *xa* is a string over ∑.

(Think of *xa* as appending a symbol to an existing string.)

Notation: The set of all strings over an alphabet ∑ is written ∑*.

Length of a string

- Many functions are defined *recursively* on the structure of strings, and many proofs are done *by induction* on the length of strings.

- *Informally*: The *length* of a string is the number of occurrences of symbols in the string (the total number of different positions at which each symbol occurs.)

- The length of string x is written as $|x|$.

Length of a string: An inductive definition

DEFINITION (length of a string)

Base: $|\varepsilon| = 0$
Induction: $|xa| = |x| + 1$

Concatenation of strings

- *Informally*: The concatenation of strings x and y over alphabet ∑ is the string formed by following x by y. It is written $x \cdot y$, or (more often) xy.

Examples

- $abc \cdot def = abcdef$
- $\varepsilon \cdot abc = abc$

Concatenation of strings: formal definition

DEFINITION (Concatentation)

The definition is recursive on the structure of the second string:

Base: $x \cdot \varepsilon = x$, if x is a string over Σ

Induction: If x and y are strings over Σ and $a \in \Sigma$ then $x \cdot (ya) = (x \cdot y)a$

Note: The parentheses are **not** symbols, they are for grouping, so $x \cdot (ya)$ is x concatenated with ya.

Languages

DEFINITION

A language over Σ is a subset of Σ^*.

Note

Of course, this omits almost everything one intuitively thinks is important about a language, such as meaning. But this definition nevertheless leads to incredibly useful and important results.

Languages: Examples

- $\varnothing$ (the empty language)
- {ε} (the language consisting only of a single empty string).
- The set of all strings with the same number of *a*'s as *b*'s.
- The set of all strings representing Java programs that compile without errors or warnings.
- The set of all theorems of number theory, in an appropriate logical notation.
- The set of all input strings for which a given Boolean function returns "true".

CARDIFF UNIVERSITY PRIFYSGOL CAERDYDD

Basic Concepts: Sequences, Functions and Relations

Sequences and tuples

- A *sequence* of objects is a list of these objects in some order, e.g., (7,21,57).
- Don't confuse sequence (7,21,57) with the set {7,21,57}!
    - Order doesn't matter in a set, but does in a sequence. (7,21,57) ≠ (21,7,57) but {7,21,57} = {21,7,57}.
    - Repetition doesn't matter in a set but does in a sequence. (7,7,21,57) ≠ (7,21,57) but {7,7,21,57} = {7,21,57}.
- Finite sequences are often called *tuples*. A sequence with *k* elements is a *k-tuple.*
- For *k* = 2 we say *ordered pair* rather than 2-tuple.

More notation about sets

- For a set A, the *power set of A*, denoted $\mathcal{P}A$ or 2^A is the set of all subsets of A.

- Example: If $A = \{0,1\}$ then $\mathcal{P}A = \{\varnothing, \{0\},\{1\},\{0,1\}\}$.

- If A and B are sets, the *Cartesian product* (or *cross product*), denoted $A \times B$, is the set of all ordered pairs (a,b) such that $a \in A$ and $b \in B$.

- Example: If $A = \{1,2\}$, $B = \{x,y,z\}$ then $A \times B = \{(1,x), (1,y), (1,z), (2,x), (2,y), (2,z)\}$

- We can also take Cartesian products of k sets $A_1 \times A_2 \times \cdots \times A_k$, which is the set of all k-tuples $(a_1, a_2, \ldots, a_k)$ with $a_i \in A_i$ for $i = 1,2,\ldots,k$.

Functions

- A *function* takes an input and produces an output.

- If *f* is a function that produces output *b* when the input is *a* then we write *f*(*a*) = *b.*

- Example: The *absolute value* function *abs*(2) = *abs*(-2) = 2.

- The set *D* of possible inputs is the *domain* of *f*, and the outputs of *f* come from a set *R* called its *range.* We write this as *f*: $D \to R$.
    - (Note: not every value in *R* may necessarily be used)

Relations

- A *predicate* or *property* is a function whose range is {TRUE, FALSE}.

- Example: property *even* outputs TRUE if the input is an even number, otherwise it outputs FALSE.

- A property whose domain is a set of k-tuples $A \times A \times \cdots A$ is called a *k-ary relation* (*on A*).

- When $k = 2$ we call it a *binary relation.*

- We often write $R(a_1,a_2,\ldots,a_k)$ to mean $R(a_1,a_2,\ldots,a_k) = \text{TRUE}$.

- For binary relations we often write aRb rather than $R(a,b)$ (*"infix" notation),* e.g., "$x < y$" for the "less than" relation $<$.

Equivalence relations

An *equivalence relation* is a special type of binary relation that captures the notion of 2 objects being equal in some feature.

DEFINITION
A binary relation *R* is an equivalence relation iff it satisfies the following 3 conditions:

1. *R* is *reflexive*, i.e., for every *a*, *aRa.*
2. *R* is *symmetric*, i.e., for every *a,b,* if *aRb* then *bRa.*
3. *R* is *transitive*, i.e., for every *a,b,c,* if *aRb* and *bRc* then *aRc.*

Equivalence relations

- Example:
  Let the relation $\equiv$ on $\mathbb{Z} \times \mathbb{Z}$ be defined by setting:

  $$(m,n) \equiv (p,q) \text{ iff } m \cdot q = p \cdot n$$

  (The set of integers, denoted by $\mathbb{Z}$, is the set of all positive and negative whole numbers $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$)

- For example, $(2,1) \equiv (4,2)$, $(3,-15) \equiv (-9,45)$

- In other words, $(m,n) \equiv (p,q)$ iff m/n and p/q are equal as fractions.

- Then $\equiv$ is an equivalence relation.

Proving Things (by Induction)

Proofs

- A *proof* is a convincing logical argument that a statement is true.

- In mathematics, proof must be airtight, beyond any doubt!

- In this module, we'll be called upon to prove quite a few things, using various standard proof techniques.

- Today we meet our first proof technique: *proof by induction.*

Proofs by induction

Consider the following mathematical statement:

For every integer i such that $i \geq 1$,

$$1 + 2 + 3 + \cdots + i = \frac{i\,(i + 1)}{2}$$

Is the statement true or false? How can we check it?

Proofs by induction

For every integer i such that $i \geq 1$,

$$1 + 2 + 3 + \cdots + i = \frac{i\,(i + 1)}{2}$$

Can try substituting different values to check it in specific instances:

$i = 1$: $1 = \frac{1 \times 2}{2}$ ✔ $i = 2$: $1+2 = \frac{2 \times 3}{2}$ ✔

$i = 3$: $1+2+3 = \frac{3 \times 4}{2}$ ✔ $i = 4$: $1+2+3+4 = \frac{4 \times 5}{2}$ ✔

But we have an infinite number of i’s. We can’t check all!!

Proofs by induction

Used to show that all elements of some infinite set have a specified property.

Example

Let $\mathbb{N} = \{1,2,3\ldots\}$ be the set of *natural numbers* (i.e., the strictly positive whole numbers), and let the property be denoted by P.

Our goal is to prove $P(k)$ holds for each natural number k (i.e., $P(1)$, $P(2)$, $P(3)$,… all hold).

Proofs by induction

2 parts to a proof by induction:

1. Base: Prove $P(1)$ holds.
2. Induction: Prove, given any $i \geq 1$, if $P(i)$ holds then so does $P(i+1)$.

the inductive hypothesis

$$\underset{\text{Base}}{P(1)} \overset{\text{Ind.}}{\Rightarrow} P(2) \overset{\text{Ind.}}{\Rightarrow} P(3) \overset{\text{Ind.}}{\Rightarrow} \cdots$$

So, after these 2 parts we know $P(i)$ holds for all i.

Proofs by induction

For every integer i such that $i \geq 1$,

$$1 + 2 + 3 + \cdots + i = \frac{i\,(i + 1)}{2}$$

Exercise (see Week 2 exercise sheet)

Prove by induction that the above statement is true.

Proofs by induction: Variations

- Can start the base case at any $k \geq 0$, rather than at $k = 1$. Then we will have shown P holds for all $i \geq k$.

- Alternative for step 2:

  Replace

  2. Induction: Prove, given any $i \geq 1$, if $P(i)$ holds then so does $P(i+1)$.

  with

  2. Induction: Prove, given any $i \geq 1$, if $P(j)$ holds for every $j \leq i$, then so does $P(i+1)$.

Example proof: Concatenation is associative

Concatenation of strings: formal definition

DEFINITION (Concatentation)

The definition is recursive on the structure of the second string:

Base: $x \cdot \varepsilon = x$, if x is a string over Σ

Induction: If x and y are strings over Σ and $a \in \Sigma$ then $x \cdot (ya) = (x \cdot y)a$

Note concatenation is defined as a *binary* function on the set of strings - it combines exactly 2 strings at a time. It takes x, y as input and produces $x \cdot y$ as output.

Concatenating 3 strings

- What if we want to concatenate *3* strings x, y, z to produce $x \cdot y \cdot z$?

- Two options:

    1. concatenate x, y first, and then concatenate the result with z, i.e., take $(x \cdot y) \cdot z$.

    2. concatenate y, z first and then concatenate x with the result, i.e., take $x \cdot (y \cdot z)$

- Intuitively the result will be the same no matter which order we concatenate, but this needs to be **proved** before we can be sure.

Concatenation is associative

Show that concatenation is *associative*, that is, for arbitrary strings x,y,z:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

[in logical notation: $\forall x,y,z$: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$]

HOW TO PROVE IT?

Recall from video 2: many properties can be proved by induction on the structure/length of strings.

Proving our result

$$\forall x,y,z\colon x\cdot(y\cdot z) = (x\cdot y)\cdot z$$

What do we take as the infinite set here?

Break the statement down into the following infinite sequence of statements:

$P(0)$: $\forall x,y,z$ such that $|z| = 0$: $\quad x\cdot(y\cdot z) = (x\cdot y)\cdot z$

$P(1)$: $\forall x,y,z$ such that $|z| = 1$: $\quad x\cdot(y\cdot z) = (x\cdot y)\cdot z$

$P(2)$: $\forall x,y,z$ such that $|z| = 2$: $\quad x\cdot(y\cdot z) = (x\cdot y)\cdot z$

and so on…

Proving our result

- Then the property $\forall x,y,z$: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ holds iff $P(i)$ holds for all $i \geq 0$.

- Proof of the latter provides a proof of the former.

- So let's prove the latter by induction.

1. Base: $\forall x,y,z$ such that $|z| = 0$: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
2. Induction: Assume $\forall x,y,z$ such that $|z| = i$: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ and then show $\forall x,y,z$ such that $|z| = i + 1$: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

For full proof, see "proof that concatenation is associative.pdf" underneath Video 5 in the Week 1 section on Learning Central

COLOUR KEY FOR PROOFS

Statements in green are things we have established as true during the proof. (The "givens")
Red statements are things we haven't established but would like to. (The "goals").

Base: $\forall x,y,z$ s.t. $|z|=0$, $x\cdot(y\cdot z)=(x\cdot y)\cdot z$

Let x,y,z be 3 strings s.t. $|z|=0$. We must show $x\cdot(y\cdot z)=(x\cdot y)\cdot z$. Since $|z|=0$ we know $z=\varepsilon$. So we must show $x\cdot(y\cdot\varepsilon)=(x\cdot y)\cdot\varepsilon$. By definition of concatenation we know $y\cdot\varepsilon=y$ and $(x\cdot y)\cdot\varepsilon=x\cdot y$ (because concatenating ε to the right of any string leaves that string unchanged). Hence $x\cdot(y\cdot\varepsilon)=x\cdot y=(x\cdot y)\cdot\varepsilon$ as required. /

Induction: Given $i\geq 0$, assume $\forall x,y,z$ st. $|z|=i$, $x\cdot(y\cdot z)=(x\cdot y)\cdot z$ and then show $\forall x,y,z$ s.t. $|z|=i+1$, $x\cdot(y\cdot z)=(x\cdot y)\cdot z$.

Let $i\geq 0$ and assume $\forall x,y,z$ s.t. $|z|=i$, $x\cdot(y\cdot z)=(x\cdot y)\cdot z$. We must show $\forall x,y,z$ s.t. $|z|=i+1$, $x\cdot(y\cdot z)=(x\cdot y)\cdot z$. Let x,y,z be s.t. $|z|=i+1$. We must show $x\cdot(y\cdot z)=(x\cdot y)\cdot z$. We know $z=w\cdot a$ for some string w (with $|w|=i$) and symbol a. So we must show $x\cdot(y\cdot(w\cdot a))=(x\cdot y)\cdot(w\cdot a)$. By definition of concatenation we know $y\cdot(w\cdot a)=(y\cdot w)\cdot a$ and $(x\cdot y)\cdot(w\cdot a)=((x\cdot y)\cdot w)\cdot a$ so we must show $x\cdot((y\cdot w)\cdot a)=((x\cdot y)\cdot w)\cdot a$. But also $x\cdot((y\cdot w)\cdot a)=(x\cdot(y\cdot w))\cdot a$ so we must show

$(x \cdot (y \cdot w)) \cdot a = ((x \cdot y) \cdot w) \cdot a$. It suffices to show $x \cdot (y \cdot w) = (x \cdot y) \cdot w$, but this holds from the inductive hypothesis and the fact $|w| = i$. ✓

Graphs

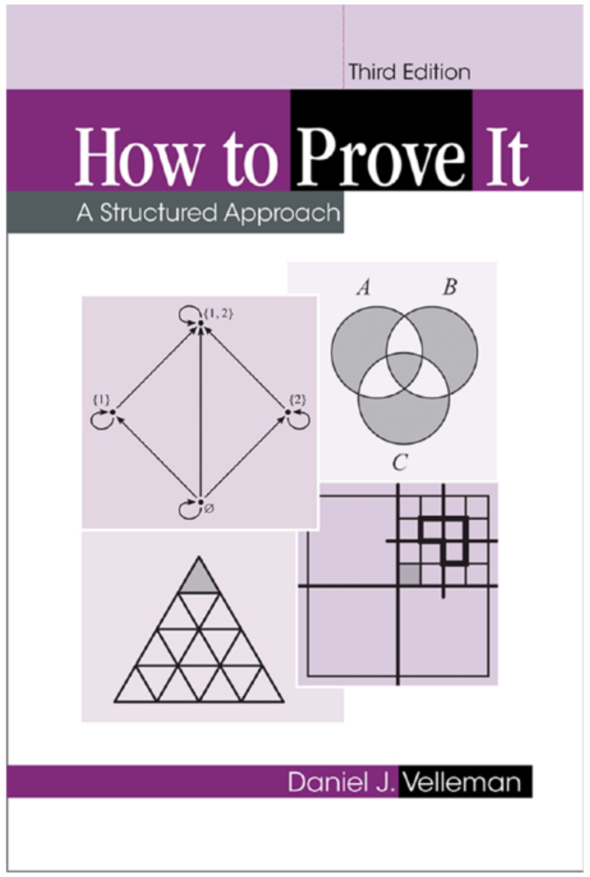
Recap

- Last week we looked at some basic concepts:
    - Languages, alphabets, strings
    - Sets, functions, relations
- Also met our first common technique for writing mathematical proofs.
    - Proof *by induction*
- We will introduce more proof techniques, e.g., proof by *contradiction*, as we go along.

Proof techniques

- A detailed textbook devoted to proof techniques is:

  Daniel Velleman,
  How To Prove It (3rd edition),
  Cambridge University Press,
  ISBN 9781108424189, 2019



- (Maybe too much detail, but worth dipping into from time to time.)
- Available to read online through library (see Reading Lists section on LC)

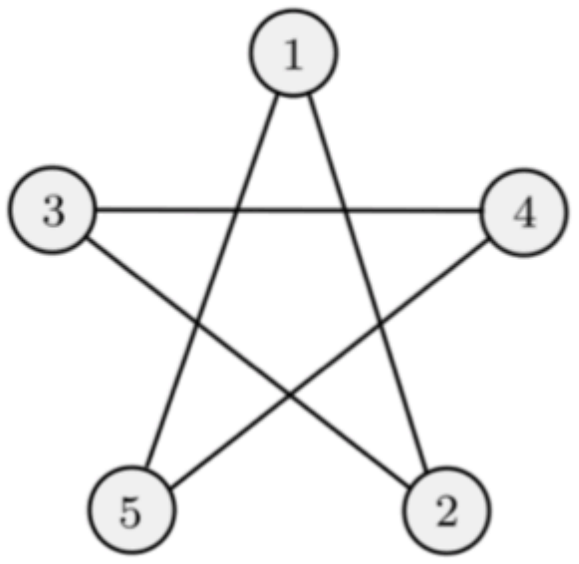
Plan for this week

- A bit of background on graphs.
- Introduction to Finite Automata

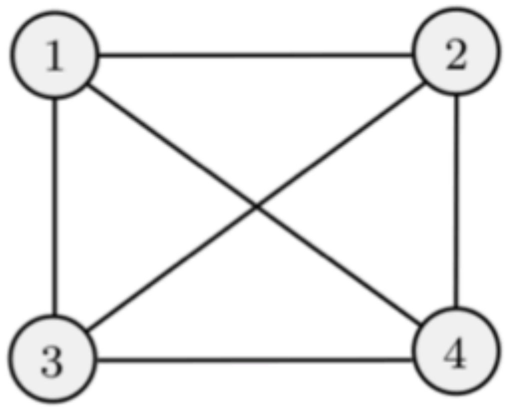
Graphs

- An *undirected graph*, or just *graph*, is a set of points with lines connecting some of the points. The points are the *nodes*, or *vertices*, and the lines are *edges.*

2 examples



(a)

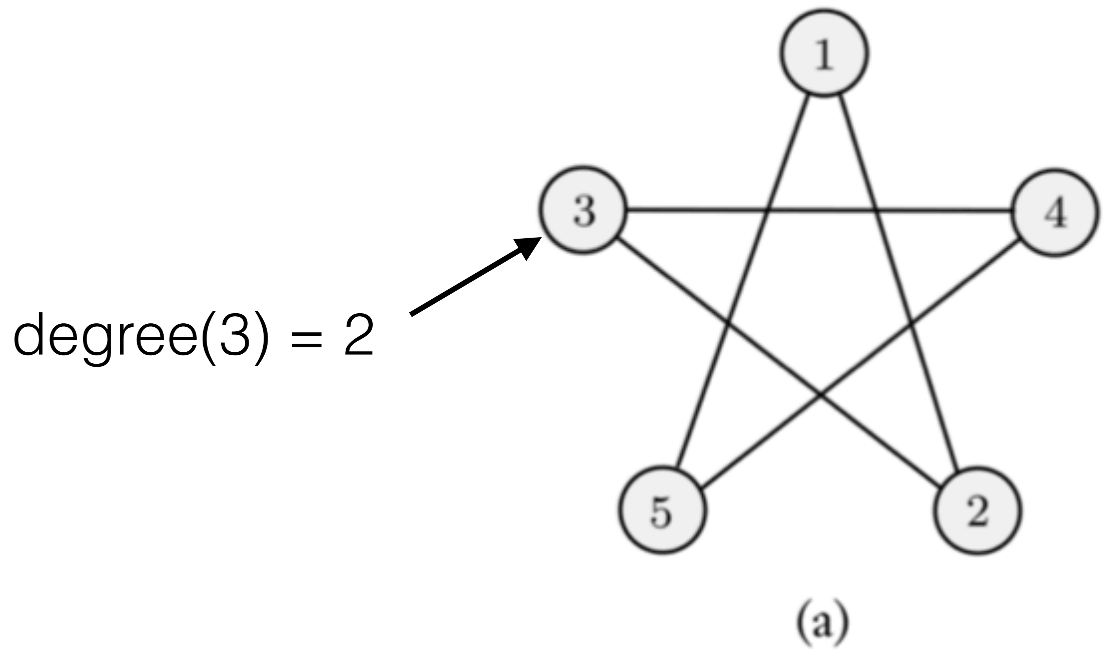


(b)

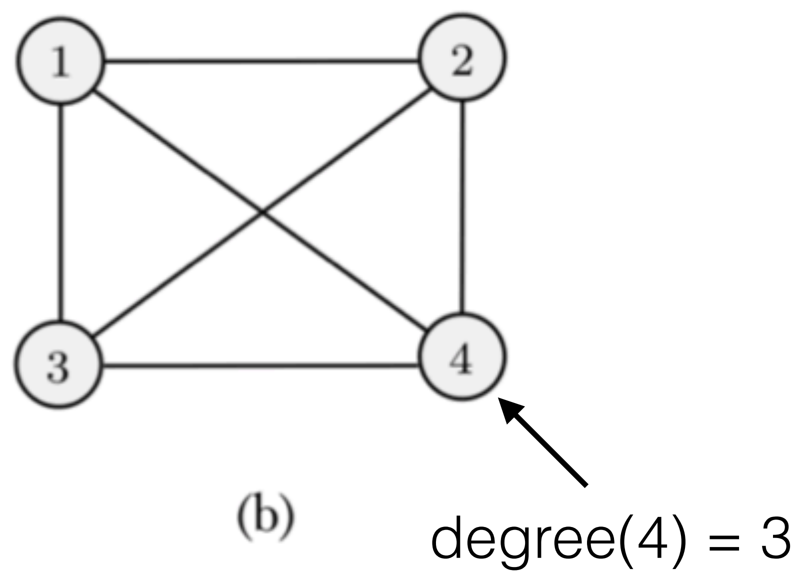
Graphs - degree

- The number of edges at a node is the *degree* of the node.

2 examples



(a)



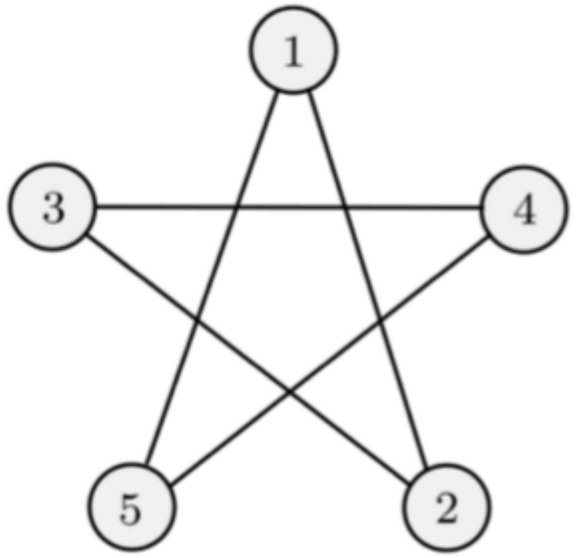
(b)

Graphs

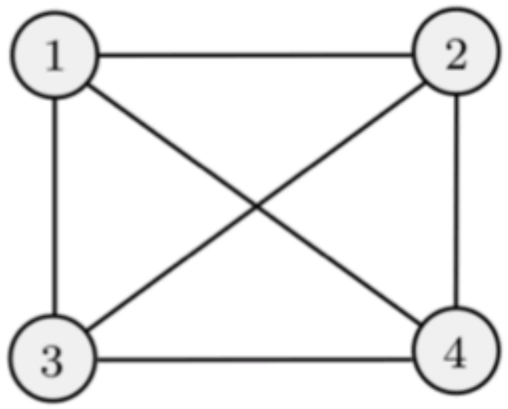
We can write graph *G* as *G* = (*V,E*), where *V,E* are the sets of vertices, edges respectively.

E.g.,
graph(a) = ({1,2,3,4,5}, {(1,2), (2,3), (3,4), (4,5), (5,1)})

(Note, no need to write include (2,1), (3,2), etc, since we are dealing with an undirected graph)



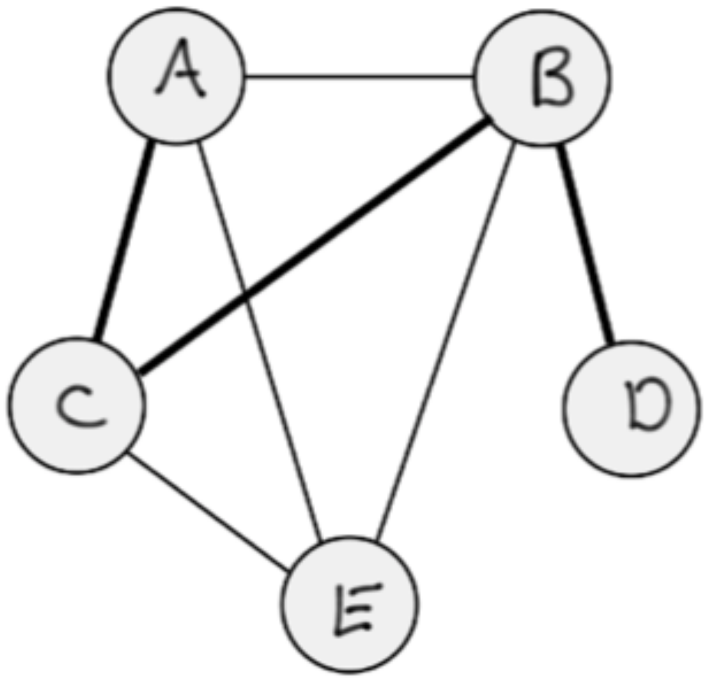
(a)



(b)

Graphs - paths

- A *path* in a graph is a sequence of nodes connected by edges.
- A graph is *connected* if every 2 nodes has a path between them.



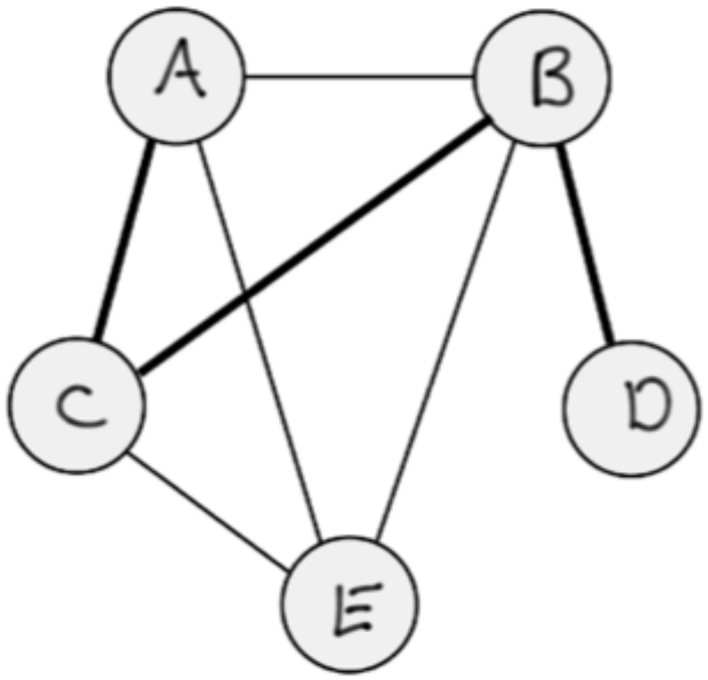
(A,C,B,D) is a path

Is this graph connected or not?

ANSWER: Yes.

Graphs - cycles

- A *cycle* is a path that starts and ends in the same node.
- A *simple cycle* is a cycle that contains at least 3 nodes and repeats only the first and last node.

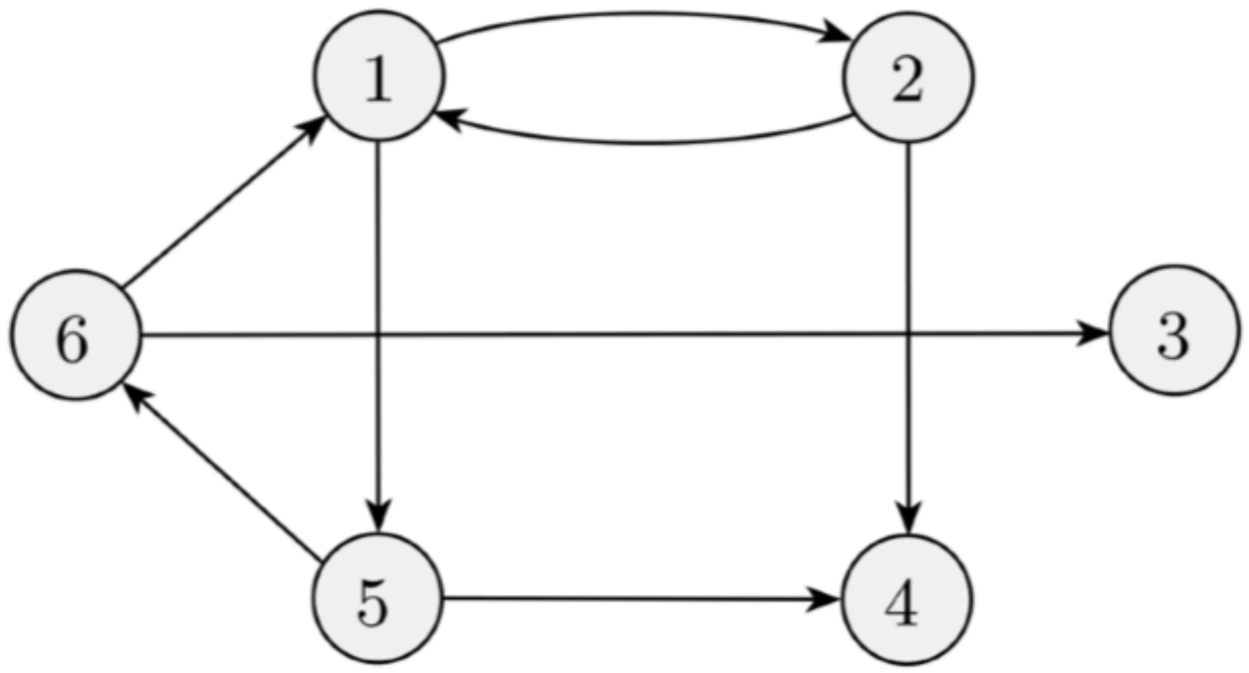


(A,C,E,A,C,E,A) is a cycle

(A,C,E,A) is a simple cycle.

Directed Graphs

- A *directed graph* has arrows instead of lines. The number of arrows pointing from a particular node is the node's *outdegree.* The number of arrows pointing to a node is its *indegree.*



outdegree(3) = 0

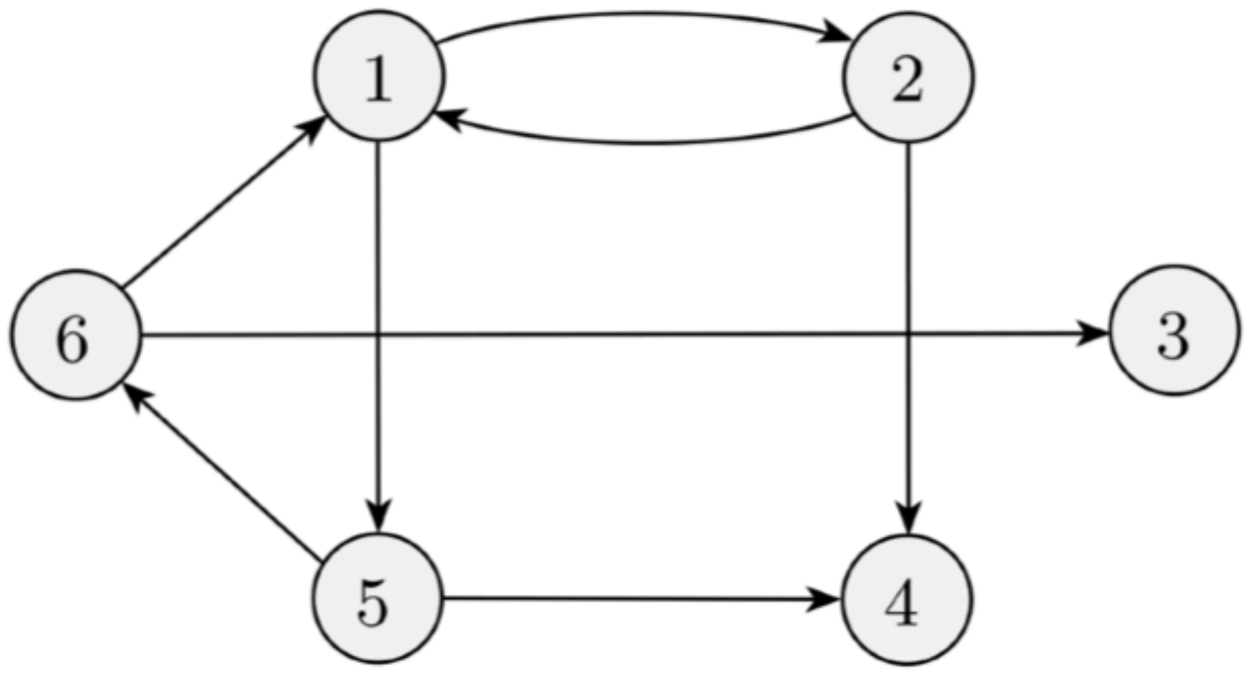
indegree(4) = 2

Directed Graphs

- We represent an edge from *i* to *j* as (*i,j*).

- Example*:* The graph below can be written

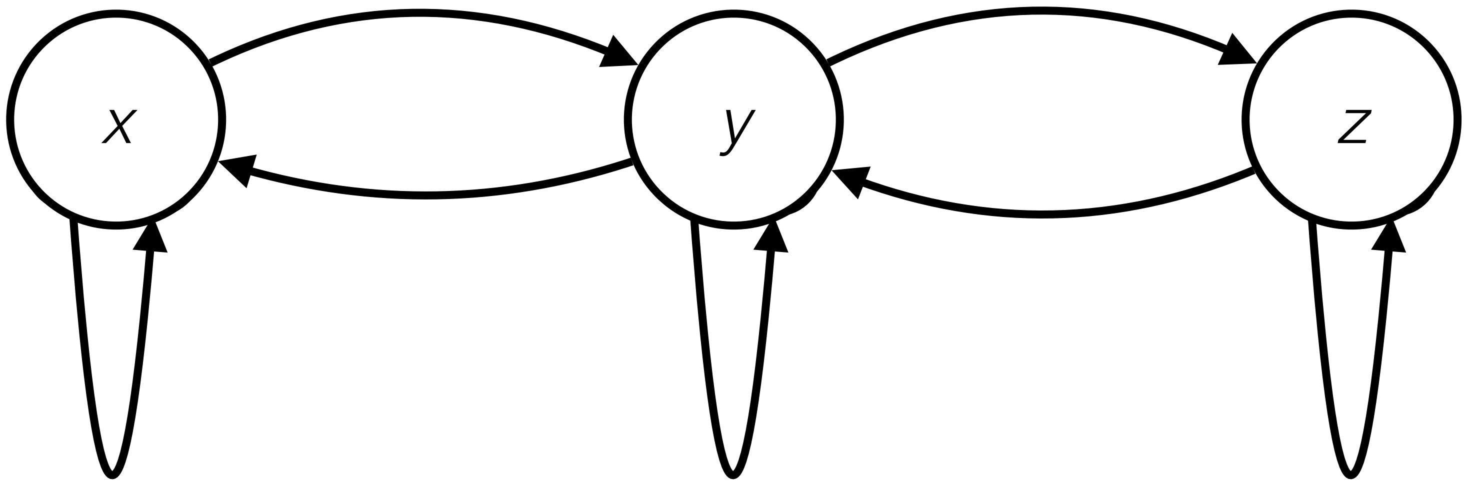
({1,2,3,4,5,6}, {(1,2),(2,1),(6,1),(6,3),(5,6),(1,5),(5,4),(2,4)})



Directed Graphs

- Directed graphs are ubiquitous in formal computer science and mathematics. They can be used to represent problems, program executions and even relations.

- Example: Let $A = \{x,y,z\}$ and let the relation R on A be specified by: xRx, yRy, zRz, xRy, yRx, yRz, zRy. Then this can be represented in graph form as follows:



symmetric + reflexive

but not transitive! (e.g., no edge (x,z))

Finite Automata

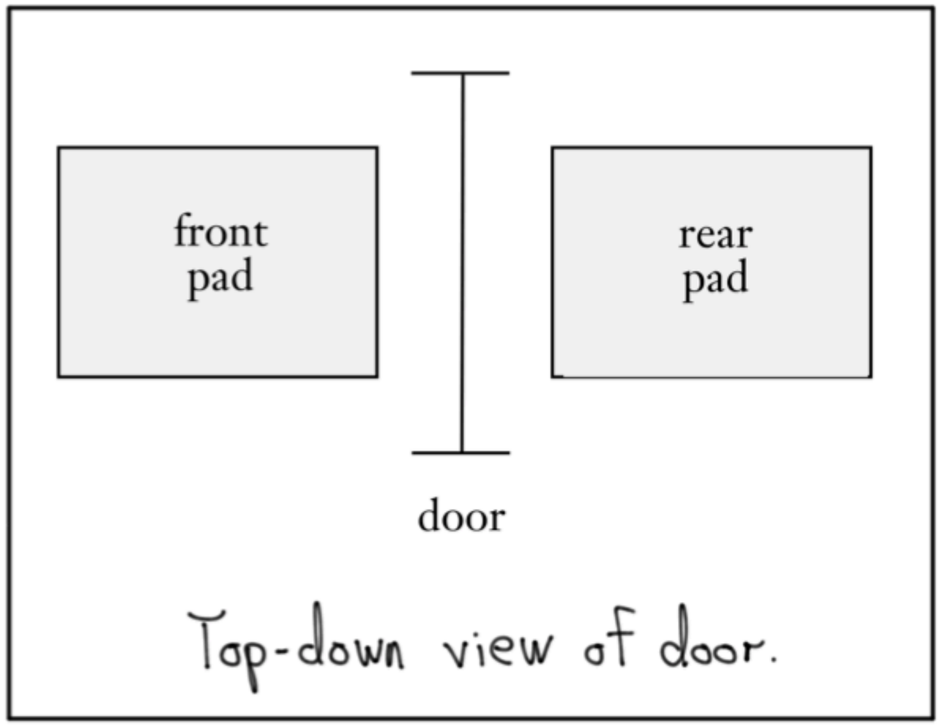
A model of computation

- Our basic question: *What is a computer?*
- What can a mathematical model of a computer look like?
- We start with the simplest model: the *finite automaton,* or *finite state machine.*

A simple example

Take a device to control an automatic door in a supermarket entrance.

- Door OPENS if someone stands on front pad (unless someone is on the rear pad at the same time).



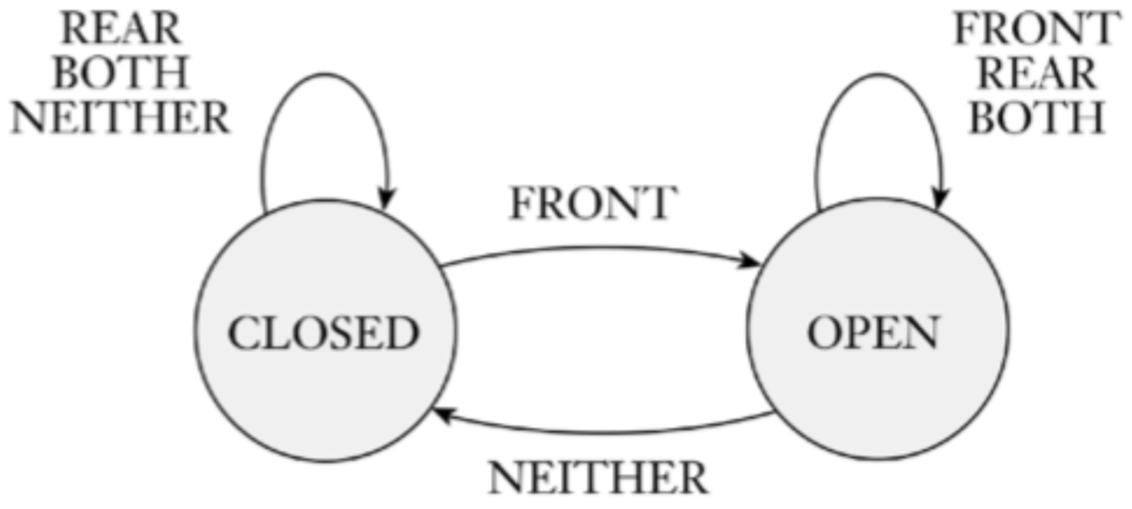
- Door then stays OPEN long enough to let them pass through.

A simple example

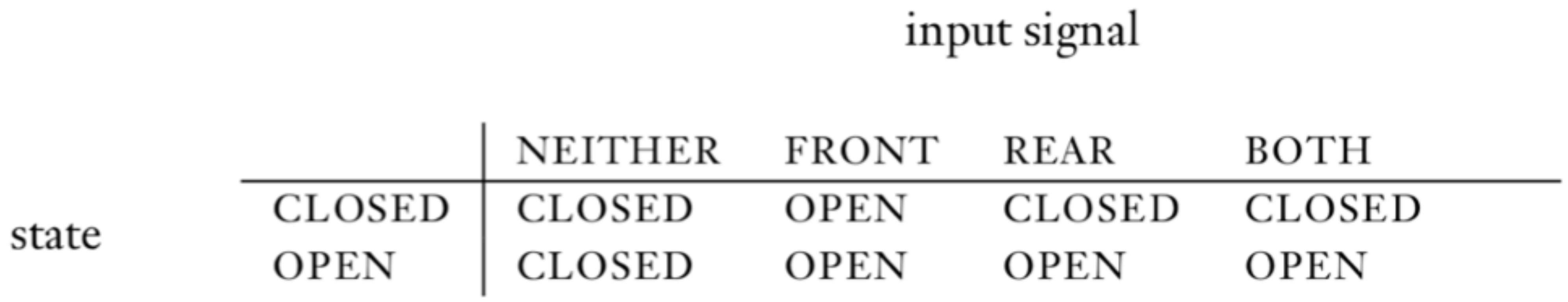
- Controller has 2 possible *states*: OPEN or CLOSED.
- 4 possible *inputs*:
    - FRONT (someone is standing on the front pad)
    - REAR (someone is standing on the rear pad)
    - BOTH (both pads have someone on them)
    - NEITHER (no one is standing on either pad)
- Controller switches from state to state, depending on the input.

A simple example

- The behaviour can be summarised by a *state diagram:*



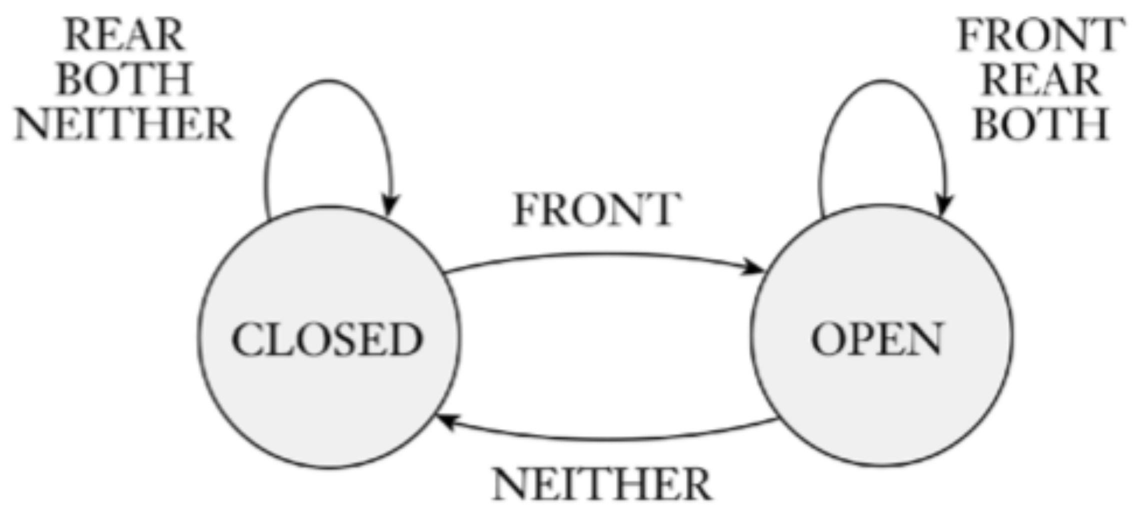
or by a *state transition table:*



input signal

state		NEITHER	FRONT	REAR	BOTH
	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

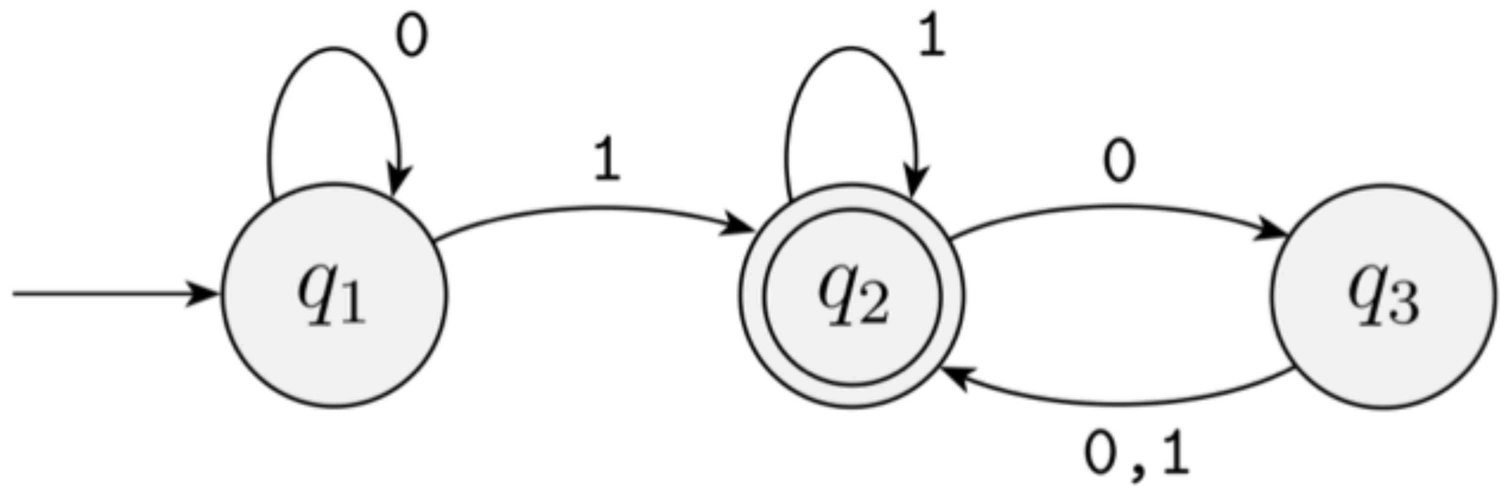
A simple example



Example: If controller starts in state CLOSED and then receives input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER then it would go through sequence of states CLOSED (start), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED.

A more general look

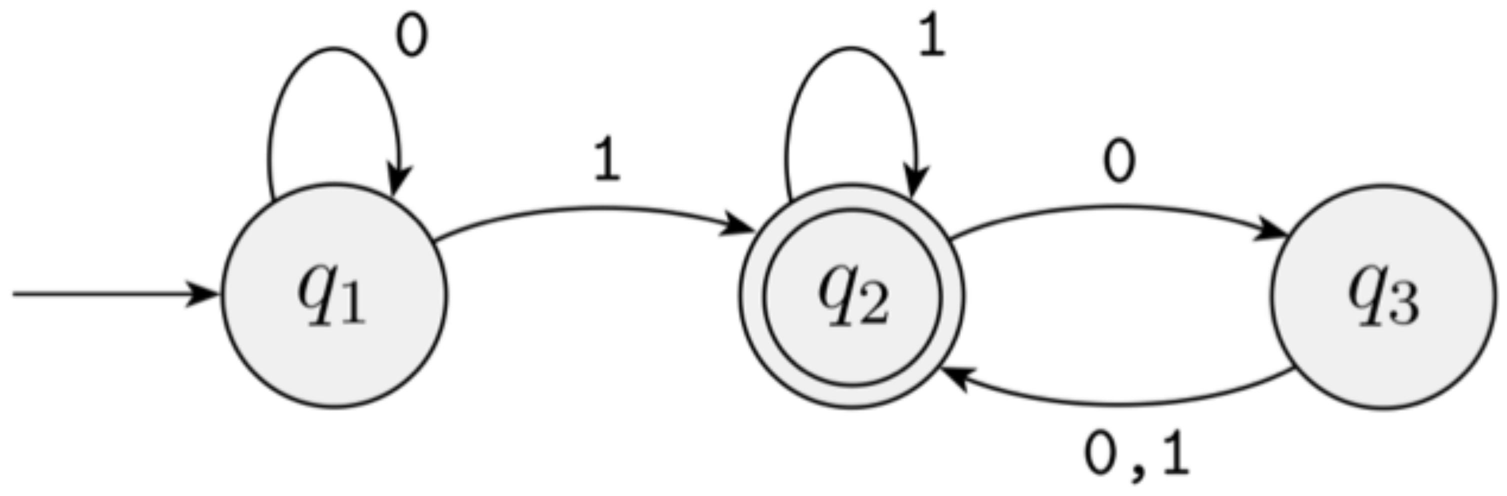
- Here is the *state diagram* of a *finite automaton* called *M.*



- *M* has
  - *states:* q_1, q_2, q_3
  - *start state:* q_1 (denoted by arrow pointing from nowhere)
  - *accept state:* q_2 (denoted by double circle)
  - *transitions* (denotes by arrows between states)

A more general look

M:

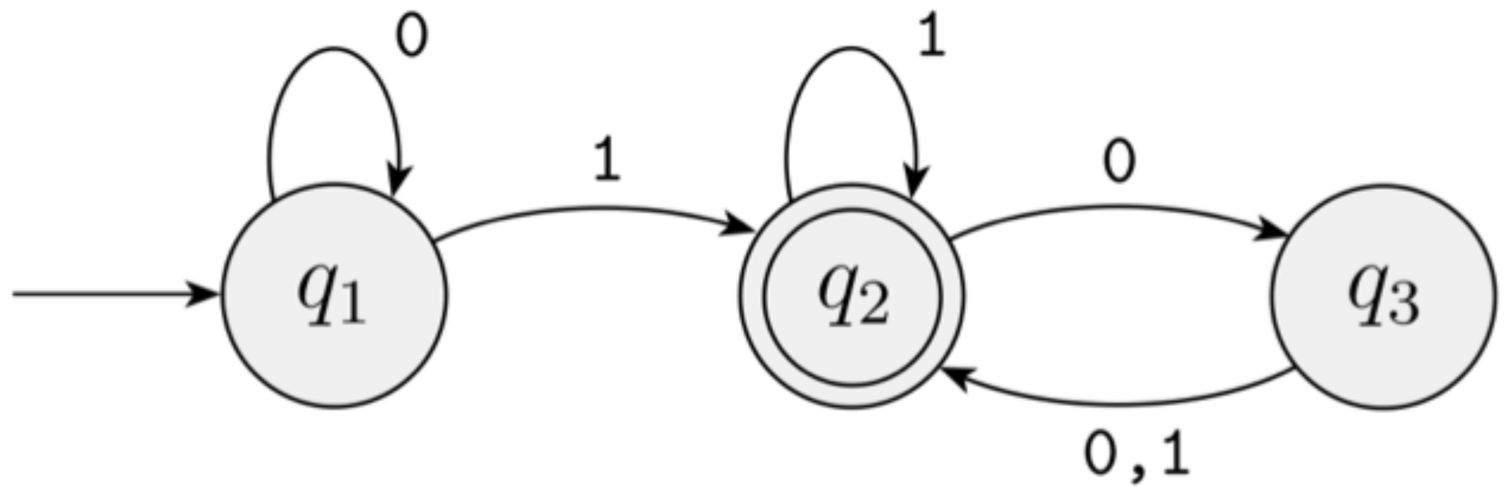


- *M* takes an *input string*, then processes it and returns an *output,* which is either *accept* or *reject.*

- Processing begins in start state, then *M* reads input string from left to right, making appropriate transition with each symbol.

A more general look

M:



Example: Input string 1101

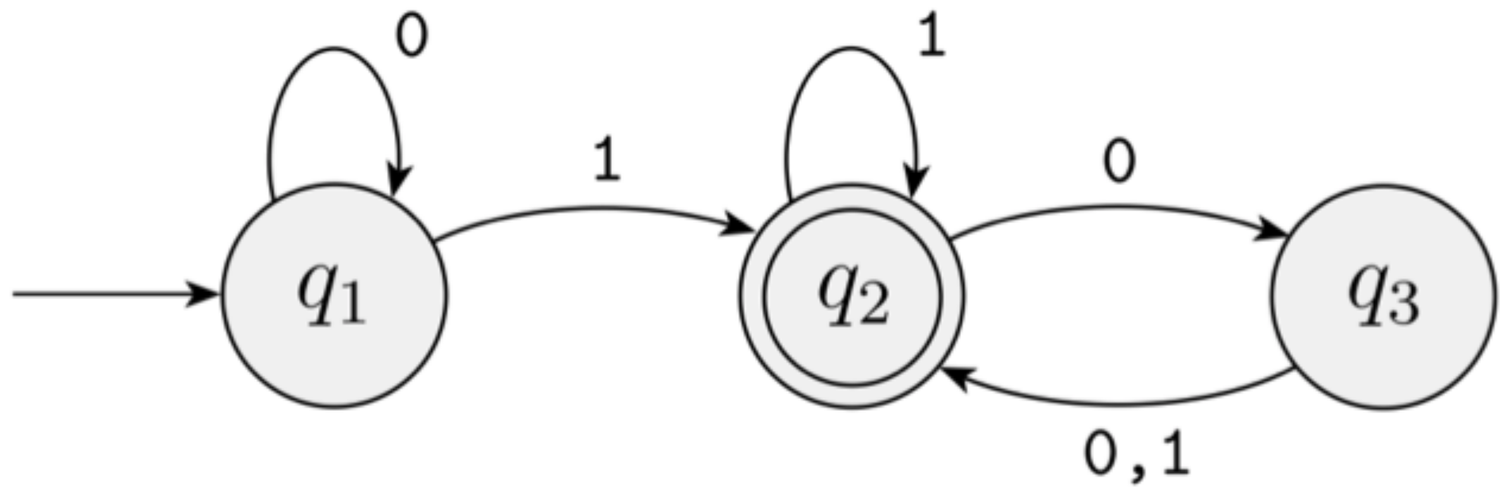
Transitions go as follows:

$$q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_2 \xrightarrow{0} q_3 \xrightarrow{1} q_2$$

String is accepted because *M* is in an accept state at the end of the input.

A more general look

M:



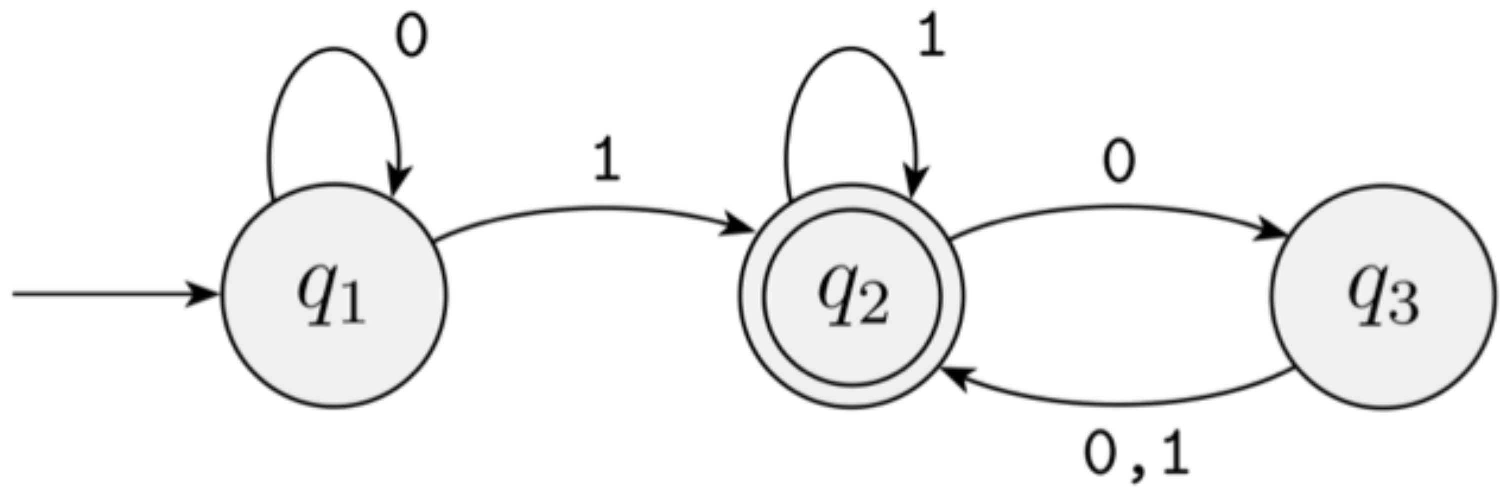
What other strings will *M* accept?

1? 101? 01101? 011101100111?

100? 010000? 101110100011100?

Finite Automata - informally

M:



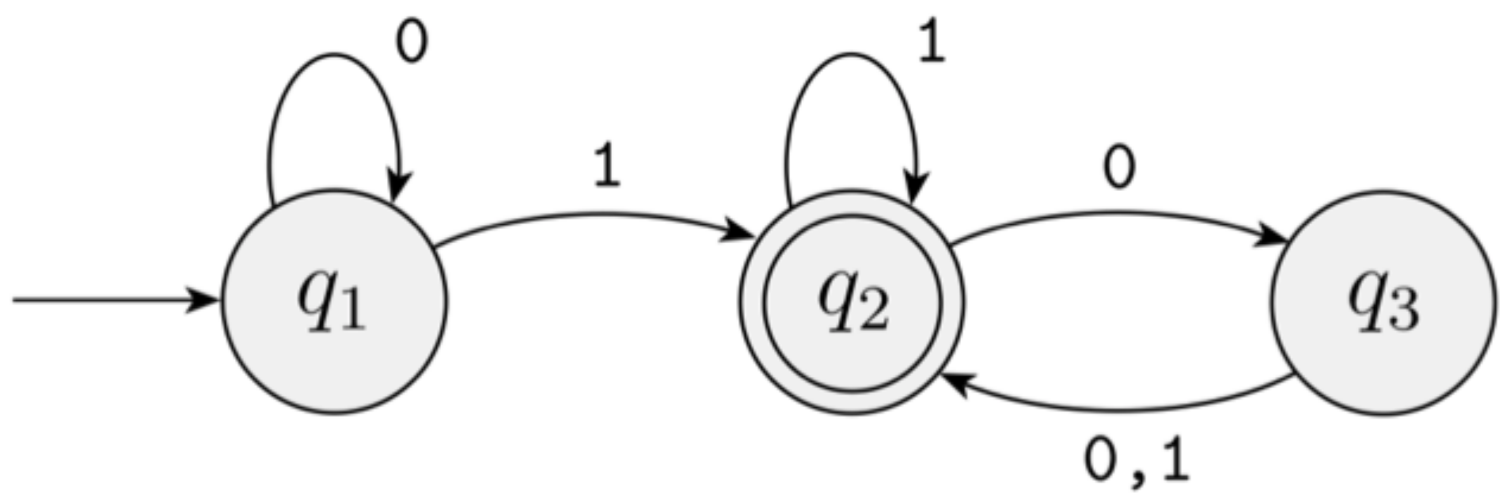
- An FA reads strings as inputs and then outputs either accept or reject.
- Example The following are all accepted:

1 101 01101 011101100111 (strings ending in '1')
100 010000 101110100011100 (strings with an even number of '0's after the last '1')

Finite Automata and Regular Languages

Finite Automata - informally

M:



- An FA reads strings as inputs and then outputs either accept or reject.
- Example The following are all accepted:

1 101 01101 011101100111 (strings ending in '1')
100 010000 101110100011100 (strings with an even number of '0's after the last '1')

Definition of a Finite Automaton

DEFINITION

A finite automaton (often abbreviated to FA) is a 5-tuple (Q,Σ,δ,q_0,F), where

1. Q is a finite set called the *states.*
2. Σ is a finite set called the *alphabet*.
3. δ is the *transition function.*
4. q_0 is the *start state*
5. $F \subseteq Q$ is the set of *accept states* (or *final states*)

[Note: FAs will also sometimes be referred to as *machines*.]

Notes on the definition

- Concerning:

  3. δ is the *transition function.*

  '$\delta(x,a) = y$' means *if FA is in state x and reads symbol a, then change to state y.*

- note δ specifies ***exactly one*** new state y for every possible pair (x,a)

- Think of (Q,Σ,δ,q_0,F) as listing the 5 *ingredients* that any particular FA must have, though each ingredient may differ from FA to FA. (E.g., different FAs M_1,M_2 might have different alphabets Σ)

Notes on the definition

- The definition allows us to be very precise about what does and does not count as being an FA.

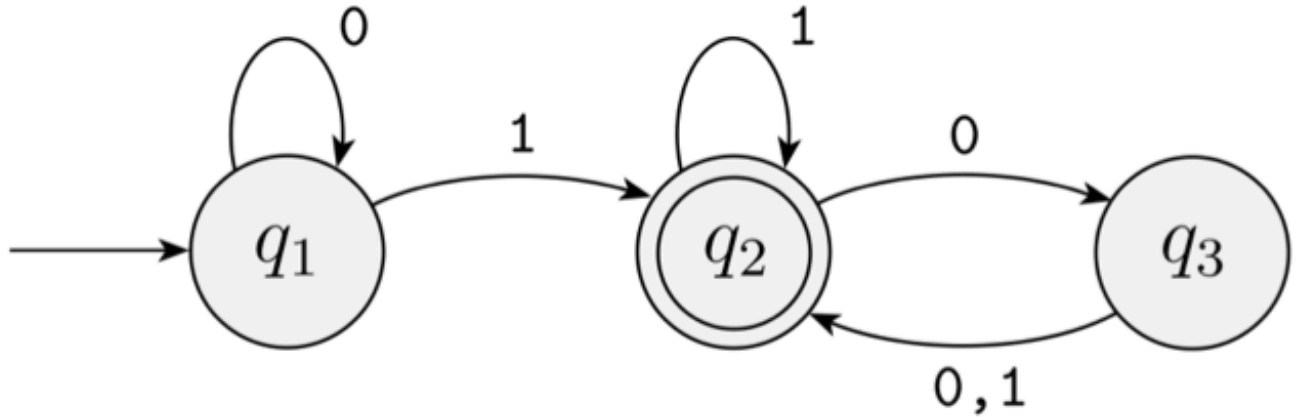
Example

Is an FA allowed to have zero accept states?

ANSWER: YES, because the set F of accept states is allowed to be *any* subset of Q, including the empty set $\varnothing$

Back to the previous example

- Recall our FA *M*:



For *M* we have:

$Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$

δ may be described as:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

q_1 is the start state, and $F = \{q_2\}$

Language recognised by an FA

DEFINITION

If A is the set of all strings that machine M accepts, we say A is the language of M, and write L(M) = A. In this case we say M recognises A.

Example

Let $A = \{w \mid w$ ends with a '1' followed by an even number of '0's$\}$

Then for M from the previous slides we have L(M) = A.

Formally defining acceptance

So far we had only an informal definition of "M accepts a string". Here comes the formal, precise definition.

DEFINITION

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA, and let $w = w_1 w_2 \cdots w_n$ be a string such that, for all i, $w_i \in \Sigma$. Then *M* accepts *w* if there exists a sequence of states $r_0, r_1, \ldots, r_n$ in Q such that:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, \ldots, n-1$
3. $r_n \in F$

Regular languages

DEFINITION
A language is called a regular language if some FA recognises it.

Example

The following language is regular:

$A = \{w \mid w$ ends with a '1' followed by an even number of '0's$\}$

because we've already seen there exists an FA M such that $L(M) = A$.

Regular languages

- The preceding definitions now suggest a number of interesting questions:

1. Which languages are regular and which are not?
2. Do there even exist languages that are not regular?
3. For a given language, how can we prove that it is regular?

SPOILER: We will soon see the answer to Q2 is YES, even for some quite simple languages. But now let's consider Q3.

Designing Finite Automata

Designing FAs

- Since a language is regular iff there is an FA that recognises it, one way to prove regularity is simply to provide such an FA.

- Example:

  Suppose $\Sigma = \{0,1\}$, and let

  $$B = \{w \mid w \text{ contains an even number of '1's}\}$$

  Is B regular?

  Let's try to design an FA to recognise it.

Hints on designing FAs

- Put yourself in the place of the machine and see how you would go about performing the machine's task.

- You read each symbol of the input string one-by-one, and at each stage you must be ready with an answer for the string seen so far.

- You need to figure out what you need to *remember* about the string as you are reading it. This will inform your choice of the set Q of states.

Designing FAs - the states

Example

$B = \{w \mid w$ contains an even number of '1's$\}$

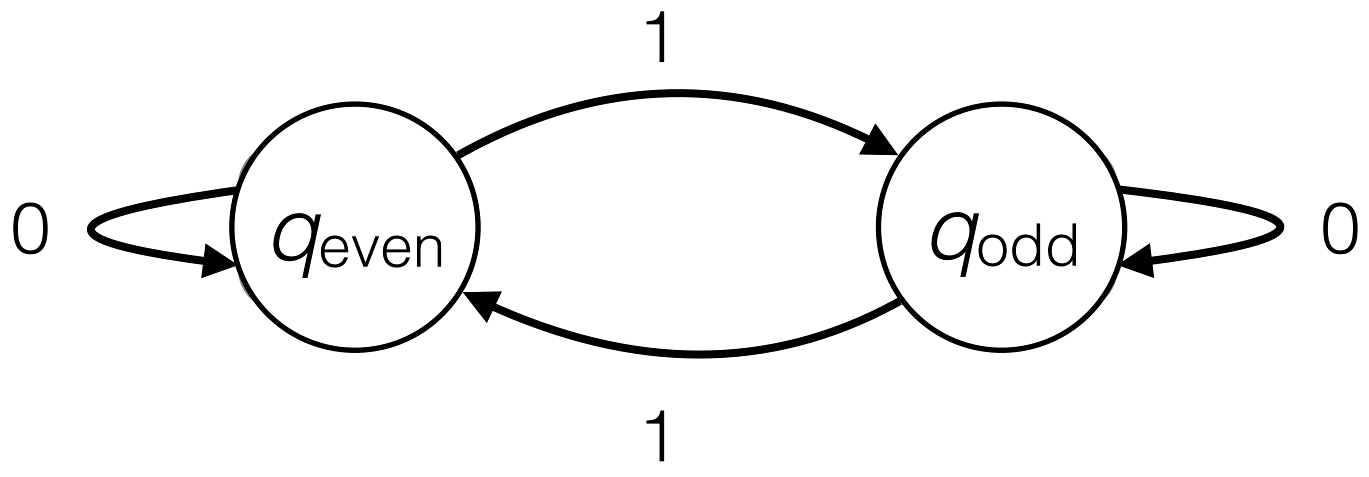
We just need to remember whether the number of '1's seen so far is even or odd. So let's have 2 states in Q, to remember this information.

Designing FAs - the transitions

Example

$B = \{w \mid w$ contains an even number of '1's$\}$

We need to add transitions (arrows) to the state diagram to capture how we change from one possibility to another upon reading a symbol.

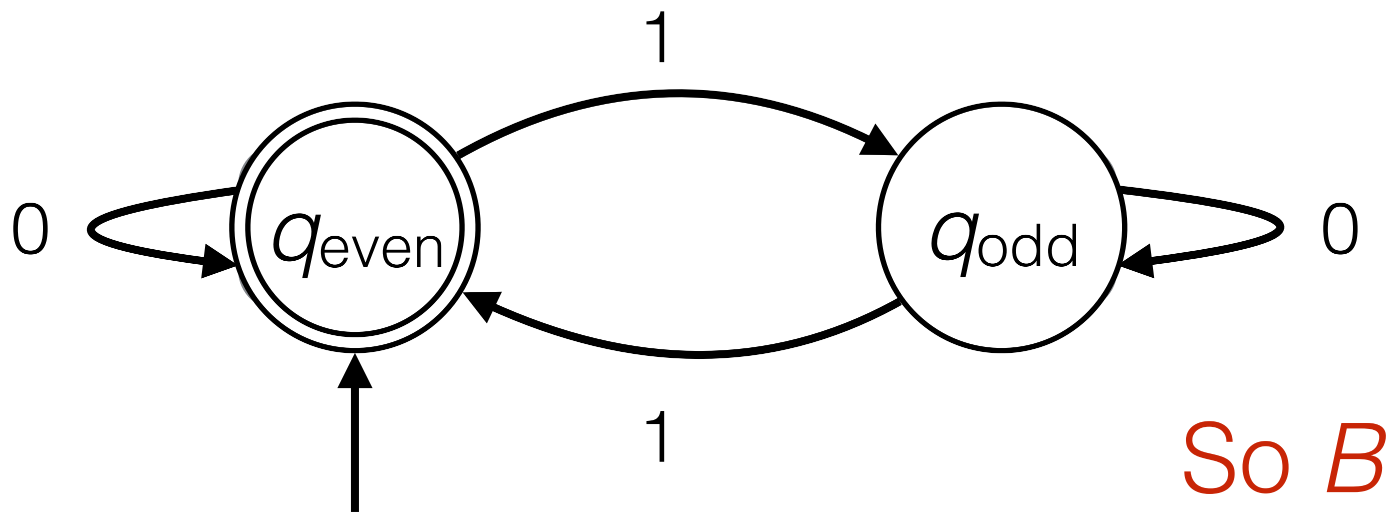

$$\begin{tikzcd}
q_{even} \arrow[r, bend left, "1"] \arrow[loop left, "0"] & q_{odd} \arrow[l, bend left, "1"] \arrow[loop right, "0"]
\end{tikzcd}$$

Designing FAs - Start and accept states

Example

$B = \{w \mid w$ contains an even number of '1's$\}$

Start state should be the state we are in when we have read zero symbols (i.e., when input is ε).
Accept state(s) are those corresponding to possibilities in which we want to accept the string.



So *B* is regular!

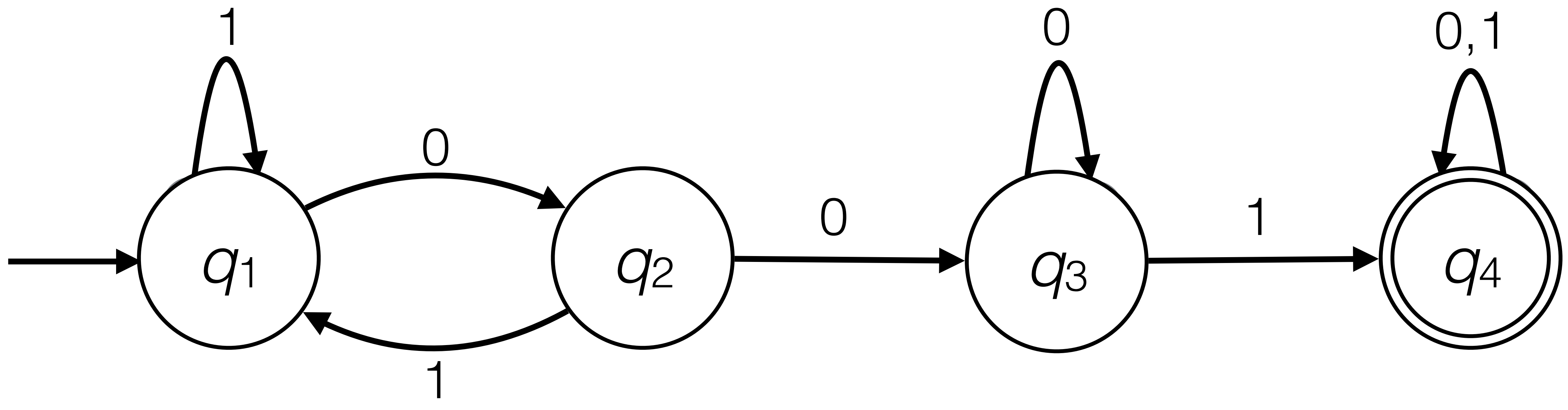
Designing FAs - Exercise

Design an FA to recognise the language (over the alphabet {0,1}) of all strings that contain 001 as a substring. (E.g. strings such as 0010, 1001, 11100111111).

Designing FAs - Exercise

Design an FA to recognise the language (over the alphabet {0,1}) of all strings that contain 001 as a substring. (E.g. strings such as 0010, 1001, 11100111111).

ANSWER:



Regular Operations

Recap

DEFINITION
A language is called a regular language if some FA recognises it.

That is, if *A* is the language in question, then it is regular iff there exists an FA *M* such that L(*M*) = *A.*

This week we explore some *properties* of regular languages, along the way developing a useful generalisation of FAs - the *nondeterministic FA* (or *NFA* for short).

Building new languages from old

DEFINITION

Let A and B be languages. We define the operations union, concatentation, and star as follows:

- Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- Star: $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

These 3 operations are sometimes known as the regular operations.

Building new languages from old

Example: Let Σ be the alphabet consisting of the standard 26 letters {a,b,...,z}. If A = {good, bad} and B = {boy, girl}, then:

- $A \cup B$ = {good, bad, boy, girl}
- $A \circ B$ = {goodboy, goodgirl, badboy, badgirl}
- A^* = {ε, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, …}

Properties of regular languages

Our first big result about regular languages concerns their interplay with the regular operations union, concatenation and star:

THEOREM

If A and B are regular languages, then so are $A \cup B$, $A \circ B$ and A^*.

Put another way: the class of regular languages is *closed under each of the 3 regular operations.*

What’s a theorem?

- ‘Theorem’ is part of the vocabulary of the mathematician. We’ve already met others such as ‘Definition’ and ‘Proof’.
    - A *theorem* is a mathematical statement that’s been proved to be true. (Sometimes *proposition* is used instead.)
- We also sometimes come across:
    - *Lemma*: same as a theorem, but usually only interesting as a stepping-stone to prove a more interesting result.
    - *Corollary*: a statement that follows immediately from a theorem, without the need for elaborate proof.

How to prove this result?

Let's take $A \circ B$ first, i.e., we assume A and B are regular, and we want to show $A \circ B$ is regular. By definition of regular language, it suffices to show there exists FA M such that $L(M) = A \circ B$. Since A, B are regular we know there exist M_1, M_2 such that $L(M_1) = A$ and $L(M_2) = B$.

Question: Can we find some way to construct M from M_1 and M_2?

Example

Let $A = \{w \mid w$ contains 001 as a substring$\}$

$B = \{w \mid w$ contains an **odd** number of ‘1’s$\}$

Quick exercise: Which of the following strings are members of $A \circ B$?

a) 100111
b) 0011011
c) 10010000
d) 0001
e) 1111

Example

Let $A = \{w \mid w \text{ contains 001 as a substring}\}$

$B = \{w \mid w \text{ contains an } \textbf{odd} \text{ number of '1's}\}$

Quick exercise: Which of the following strings are members of $A \circ B$?

a) 10011⋮1 YES

b) 001⋮1011 YES

(or 001101⋮1)

c) 10010000 NO

d) 0001 NO

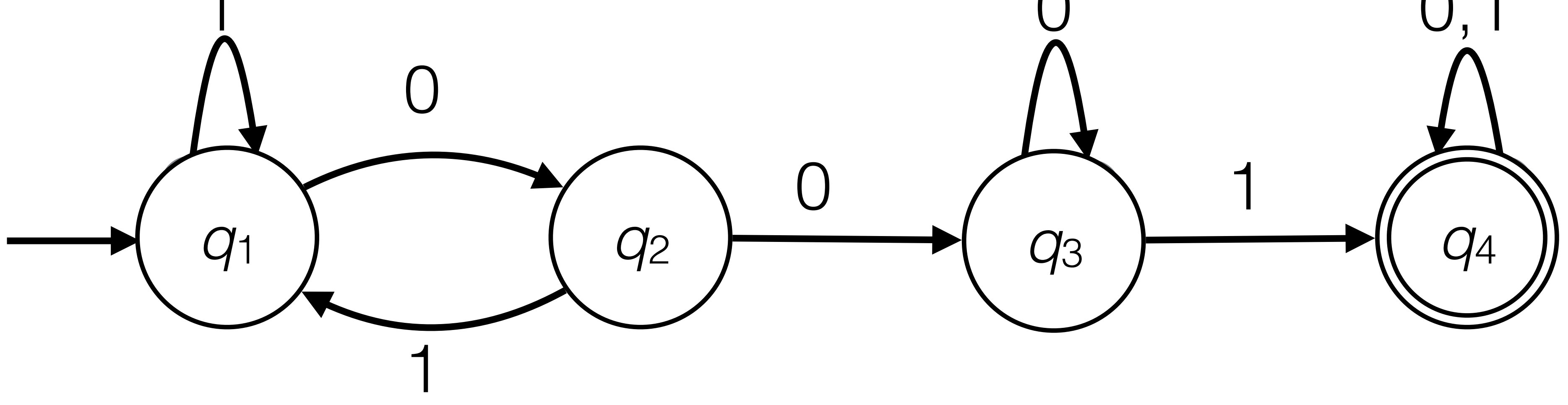
e) 1111 NO

Example

Let A = {w | w contains 001 as a substring}

B = {w | w contains an **odd** number of ‘1’s}

We already constructed an FA M_1 such that L(M_1) = A last lecture.



$$\begin{tikzcd}
\arrow[r] & q_1 \arrow[loop above, "1"] \arrow[r, bend left, "0"] & q_2 \arrow[l, bend left, "1"] \arrow[r, "0"] & q_3 \arrow[loop above, "0"] \arrow[r, "1"] & q_4 \arrow[loop above, "{0,1}"]
\end{tikzcd}$$

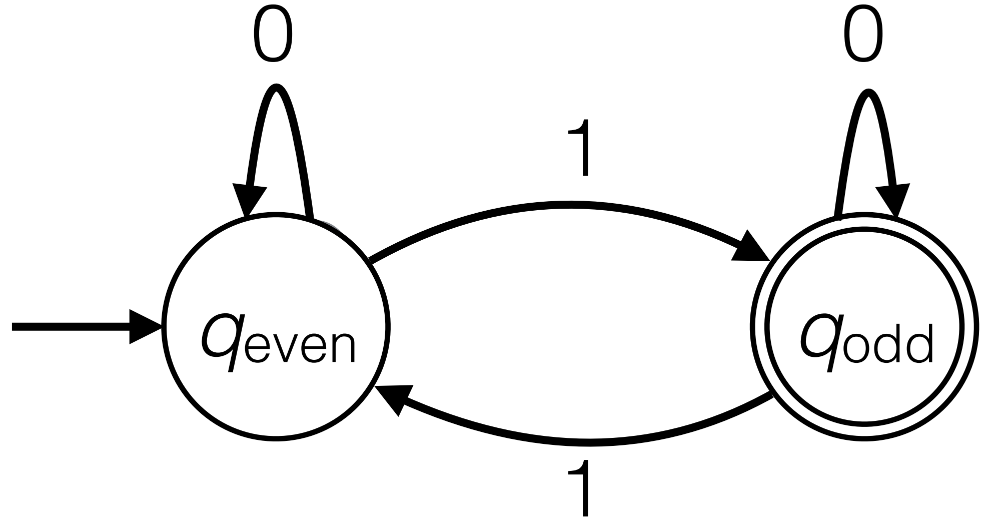
M_1

Example

Let A = {w | w contains 001 as a substring}

B = {w | w contains an **odd** number of ‘1’s}

We can obtain an FA M_2 such that L(M_2) = B via a slight modification to the FA we saw last week that recognises the set of strings with **even** number of ‘1’s (switch final state to q_{odd} rather than q_{even})



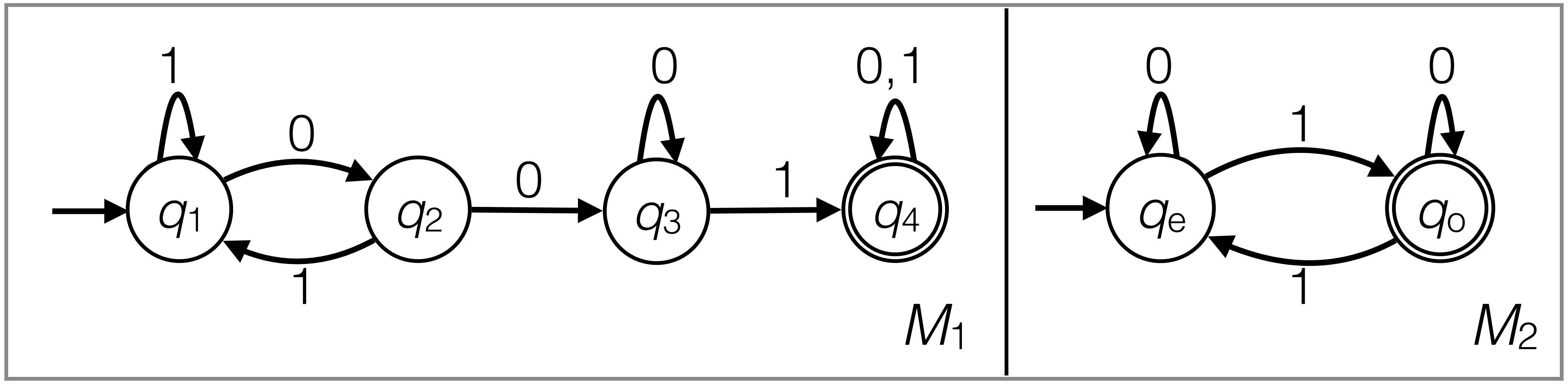
M_2

Example

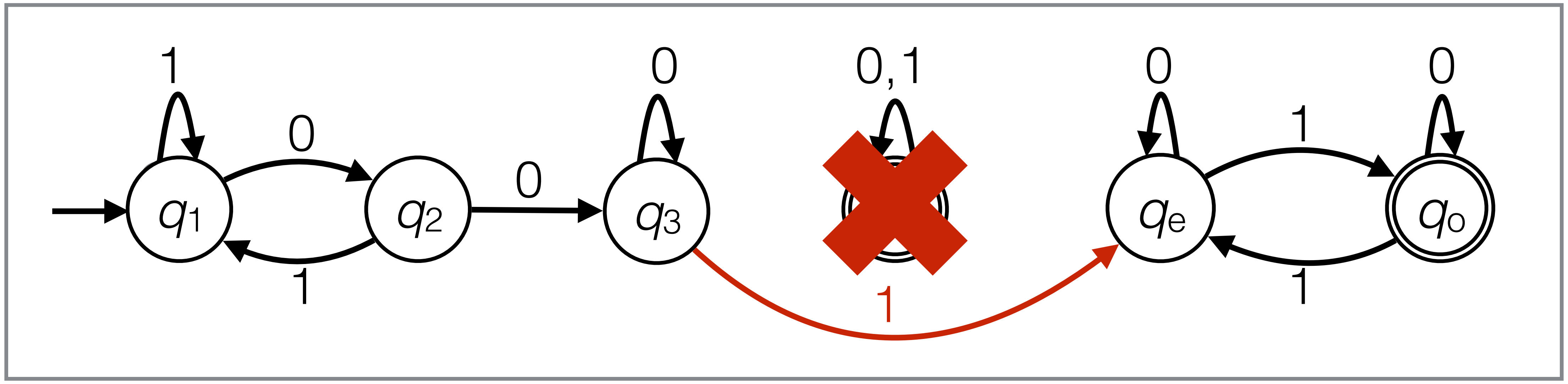
Let A = {w | w contains 001 as a substring}

B = {w | w contains an **odd** number of ‘1’s}

Can we combine M_1 and M_2 into a single FA that recognises $A \circ B$?



Example



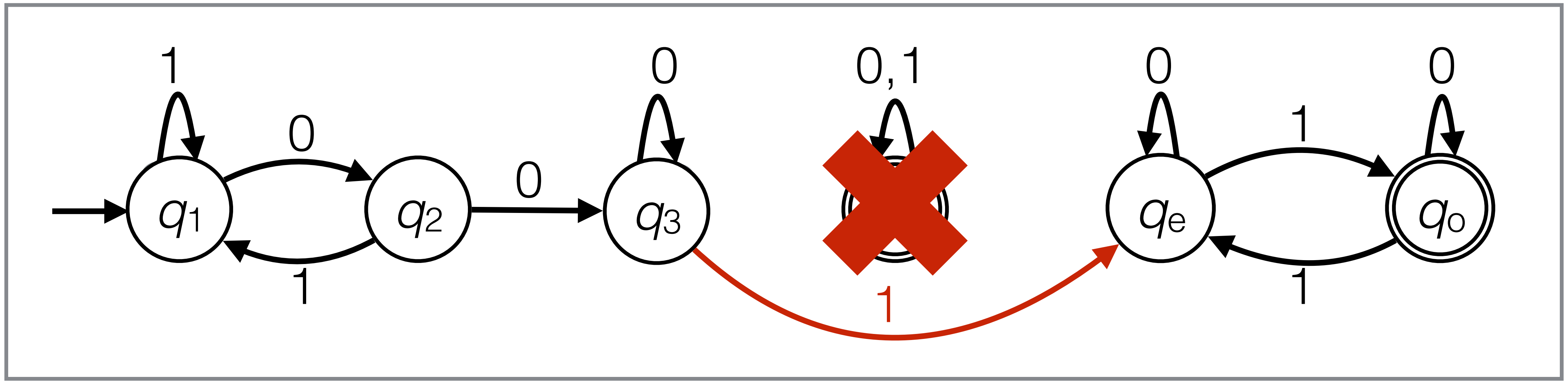
FIRST IDEA

Create one big FA M out of M_1, M_2 by:

(i) setting the start and accept states of M to be the start state of M_1 and the accept state of M_2 respectively.

(ii) removing the accept state q_4 of M_1 and diverting all arrows leading to q_4 into the start state q_e of M_2.

Example



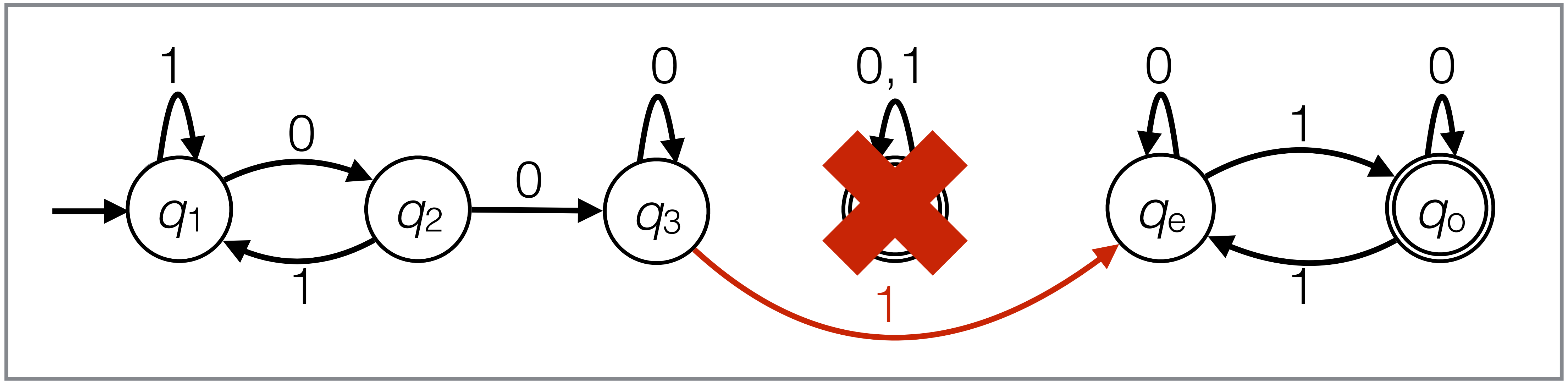
Does this machine recognise $A \circ B$?

We have, for example, 0011 $\in A \circ B$, and 0011 accepted in M. 😃

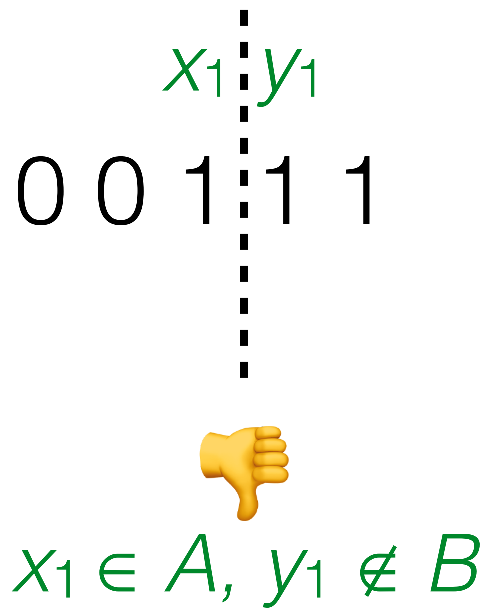
But we also have, e.g., 00111 $\in A \circ B$ and 00111 rejected in M. 😔

So the answer is NO.

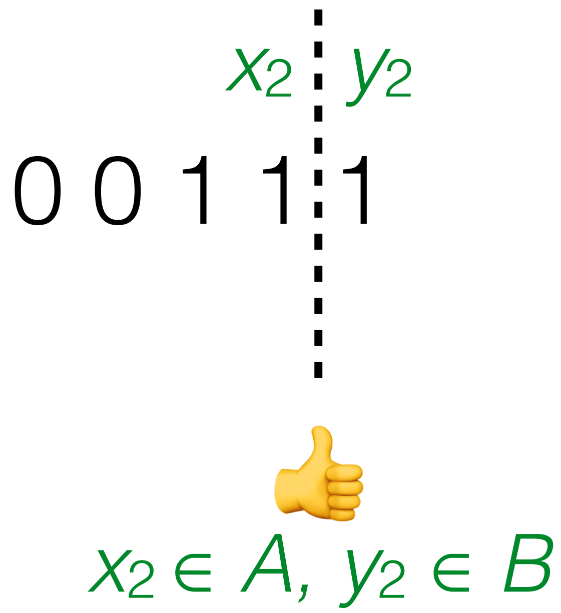
Example



The problem is that we can't predict during run-time where the part of the string accepted by M_1 *ends*.



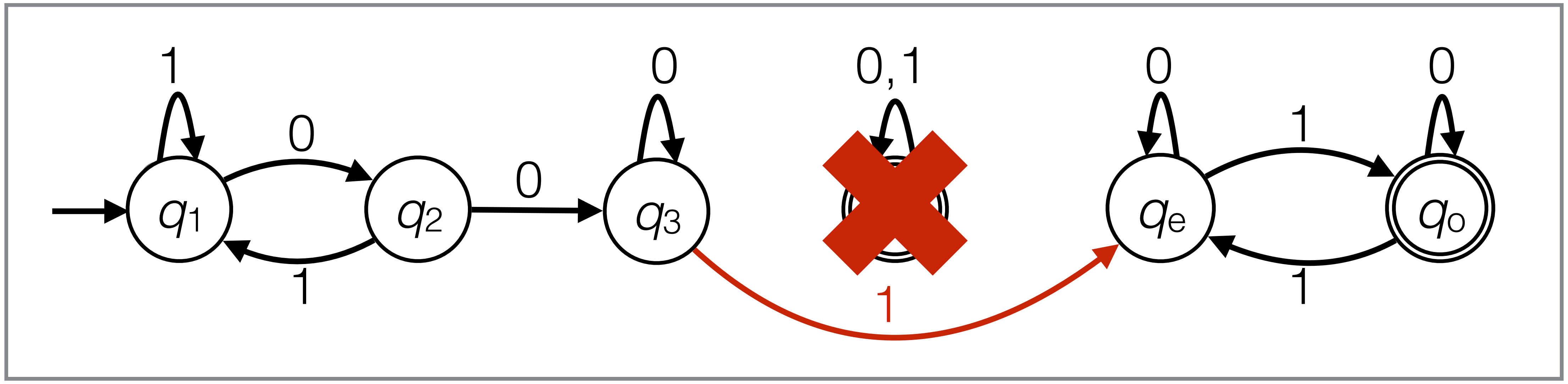
$x_1 \in A$, $y_1 \notin B$



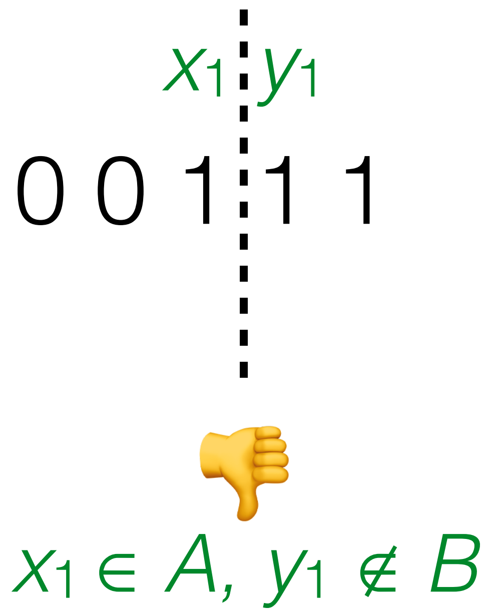
$x_2 \in A$, $y_2 \in B$

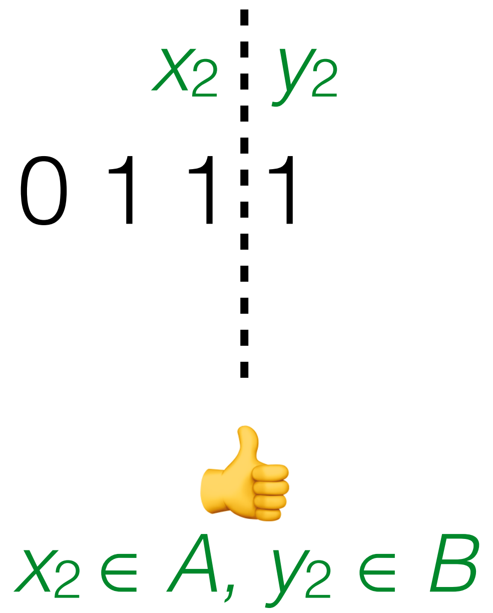
Nondeterministic Finite Automata

Example

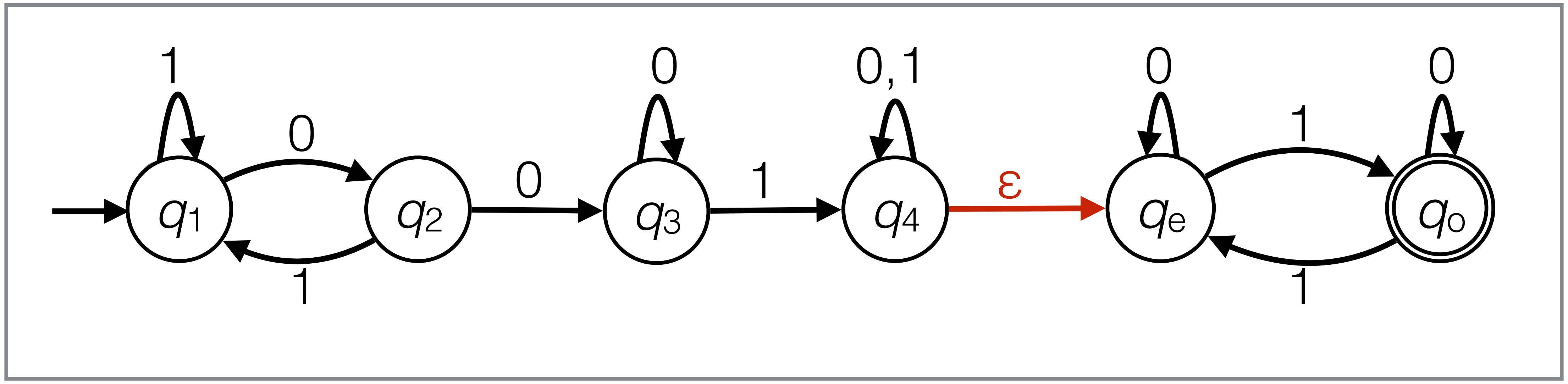


The problem is that we can't predict during run-time where the part of the string accepted by M_1 *ends*.





Introducing Nondeterministic Finite Automata

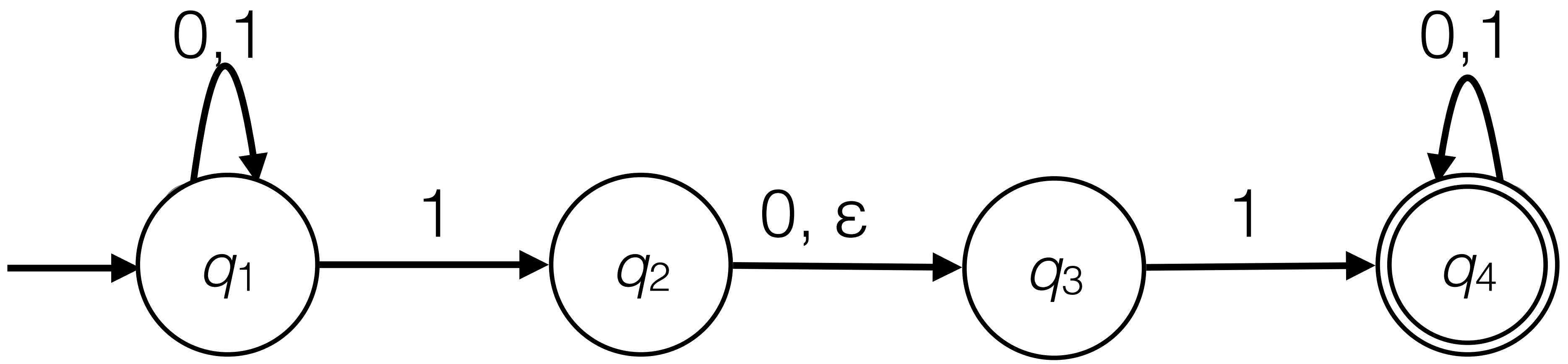


- In fact there *is* a way to combine M_1 and M_2 into a machine that recognises $A \circ B$, but it requires us to consider a new type of machine - nondeterministic finite automata (NFAs).
- NFAs essentially provide a way to do parallel computation.

Terminology

- NFAs are a *generalisation* of FAs. That is, every FA is an NFA by their definitions (see later for the formal definition of NFA).
- To bring out the contrast with NFAs, we will now refer to FAs as deterministic finite automata (DFAs).

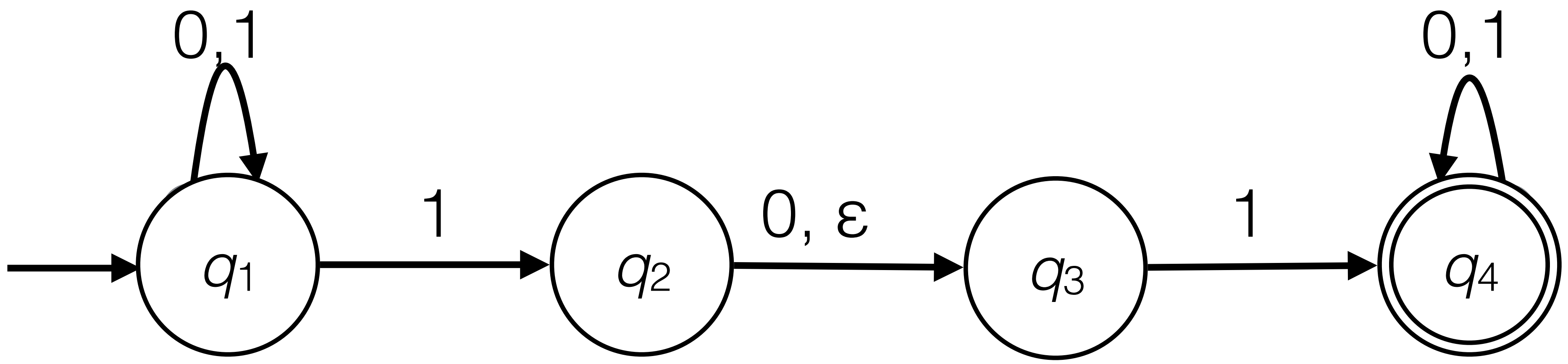
Differences between NFAs and DFAs



FIRST DIFFERENCE

- DFAs always have exactly **one** exiting transition for each symbol in ∑.
- NFAs may have zero (e.g., q_3 with symbol 0), one or more than one (e.g., q_1 with symbol 1) exiting arrows for each symbol in ∑.

Differences between NFAs and DFAs



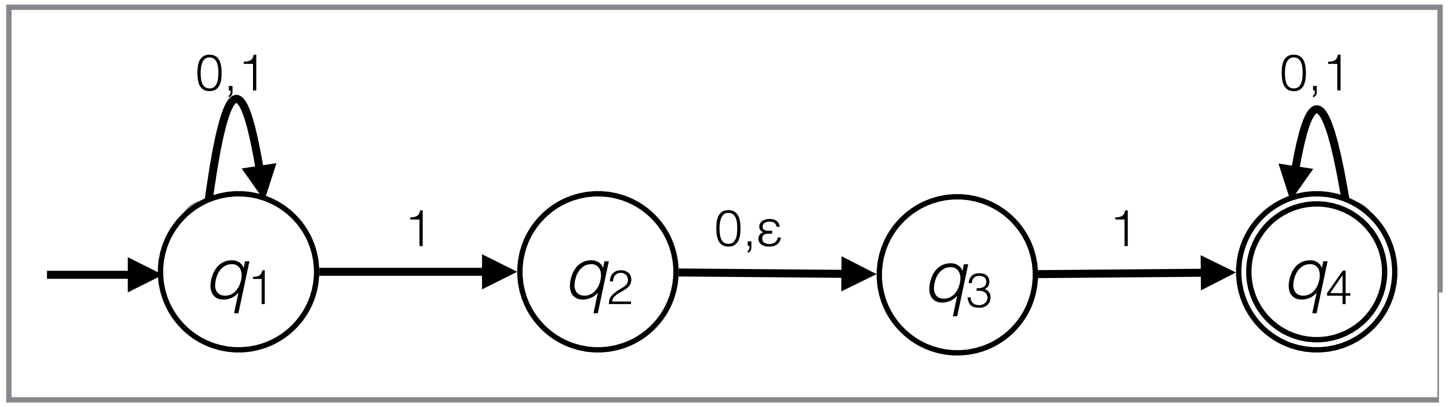
SECOND DIFFERENCE

- Unlike DFAs, NFAs can have transition arrows labelled with the symbol ε standing for 'blank' (e.g., q_2).

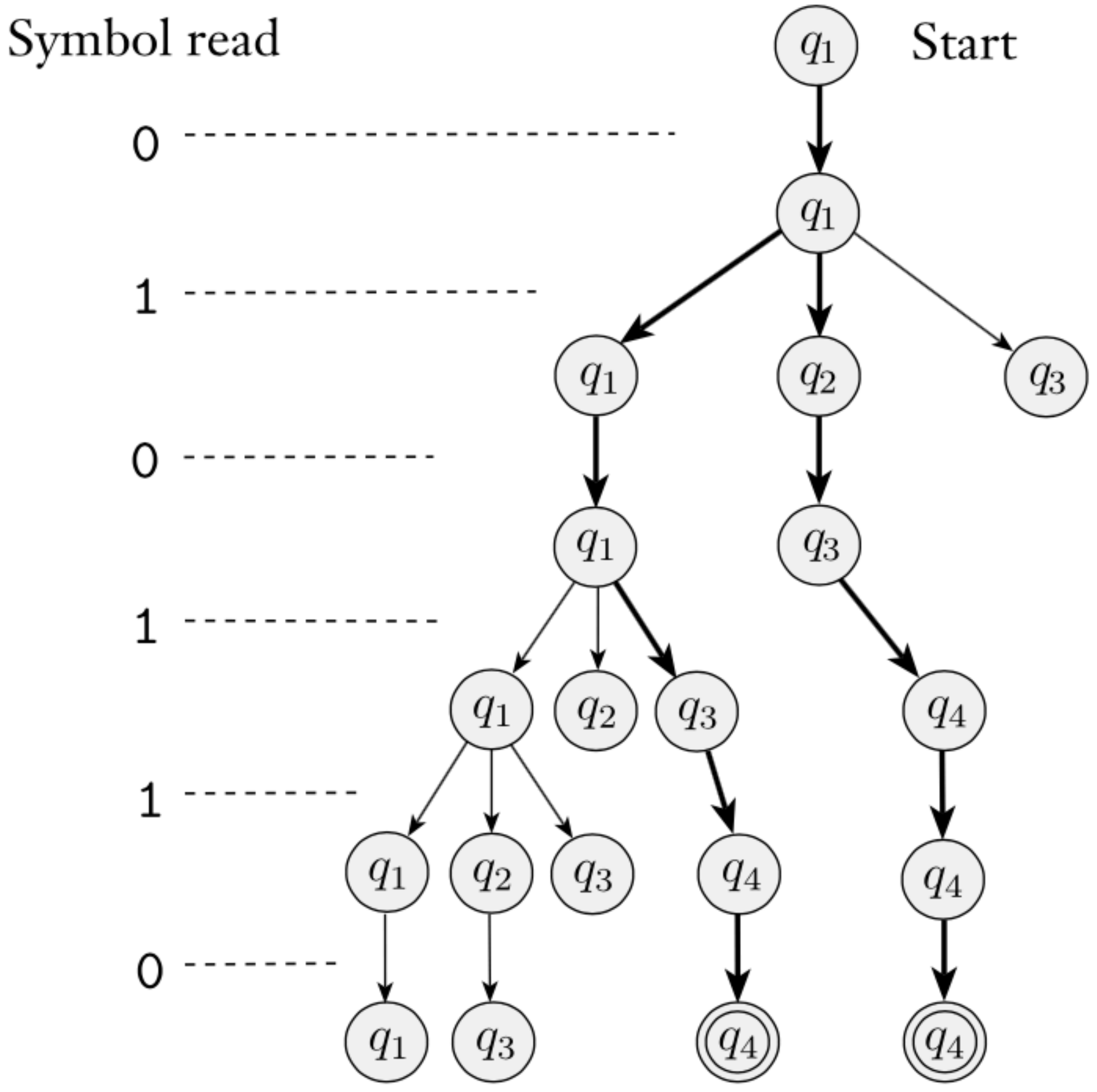
How does an NFA compute?

- If we reach a state with multiple ways to proceed (e.g., q_1 on previous slide), then machine splits into multiple copies of itself and follows all possibilities in parallel.
- If next input symbol doesn't appear on any arrow (e.g., q_3 and symbol 0) then that copy of the machine "dies".
- If we reach a state with an ε-arrow then again split into multiple copies - one following each ε-arrow and one staying at current state.
- If at least one branch of computation ends in accept state then the entire computation accepts.

Example



Computation on input 010110

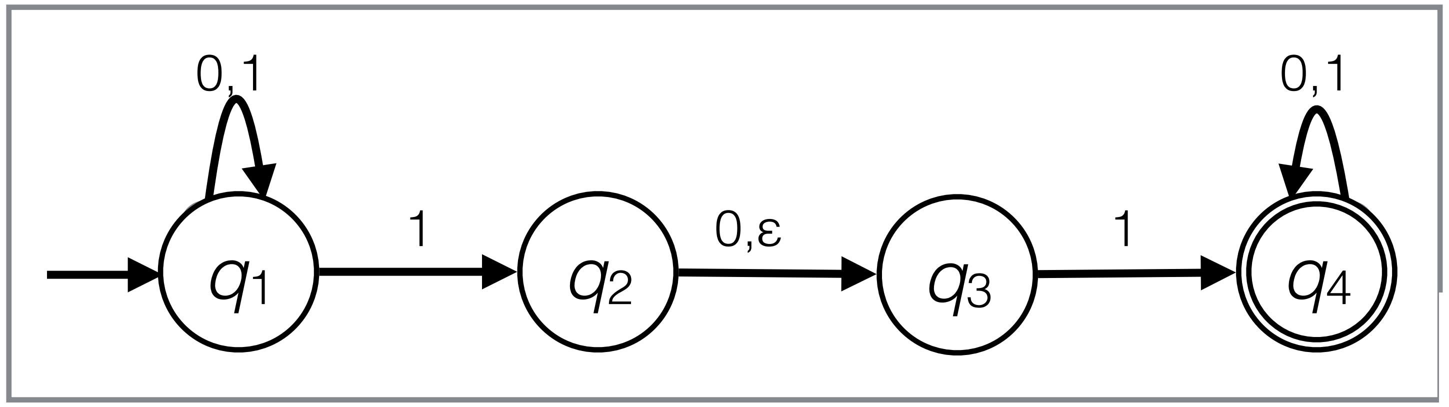


Can visualise the computation as a tree† $\longrightarrow$

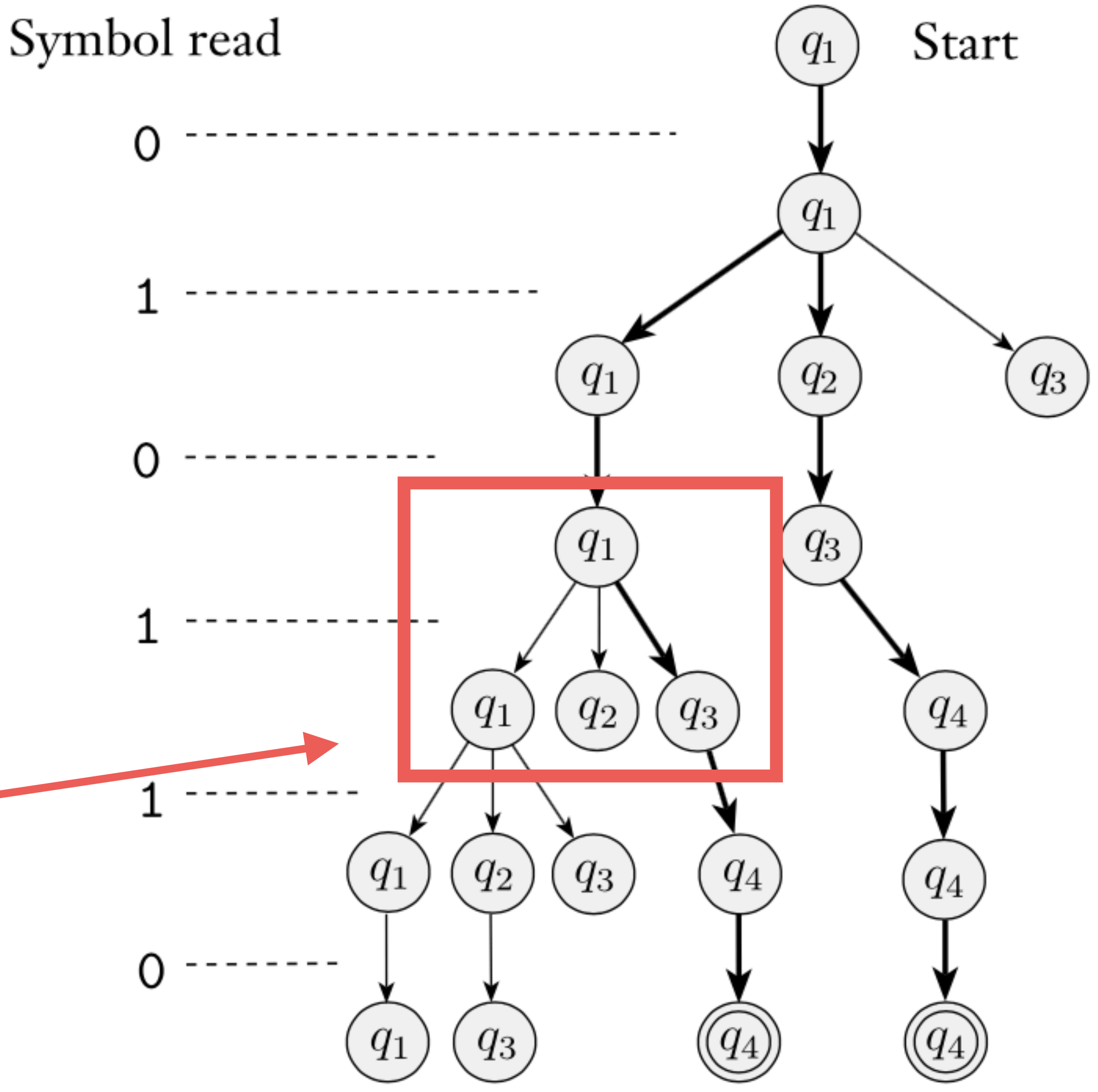
At least one branch ends in an accept state (q_4) so string is accepted

† A tree is a connected graph with no simple cycles

Example

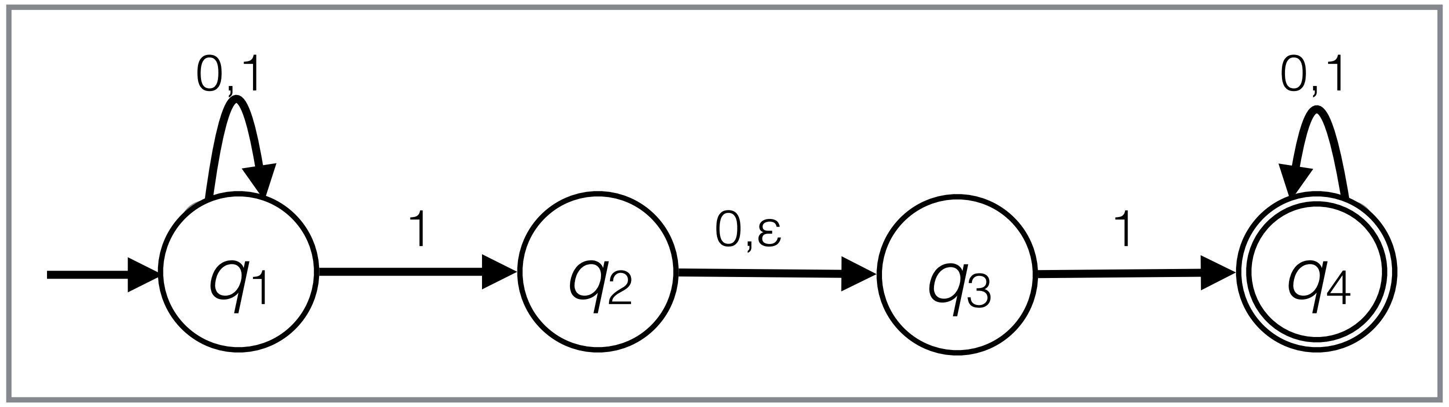


Computation on input 010110



In this visualisation tree the children of each node are the possible nodes reachable from that node after reading the given symbol. E.g., after reading 1 in q_1 the state can be q_1, q_2 or q_3.

Exercise


$$\begin{tikzcd}
\arrow[r] & q_1 \arrow[loop above, "{0,1}"] \arrow[r, "1"] & q_2 \arrow[r, "{0,\varepsilon}"] & q_3 \arrow[r, "1"] & q_4 \arrow[loop above, "{0,1}"]
\end{tikzcd}$$

Questions

(i) Is string 010 accepted by this NFA?

(ii) What is the language recognised by this machine?

ANSWERS

(i) NO (see diagram below Video 2, Week 3 on Learning Central)

(ii) $\{w \mid w$ contains either 11 or 101 as a substring$\}$

NFAs: The Formal Definition

Definition of an NFA

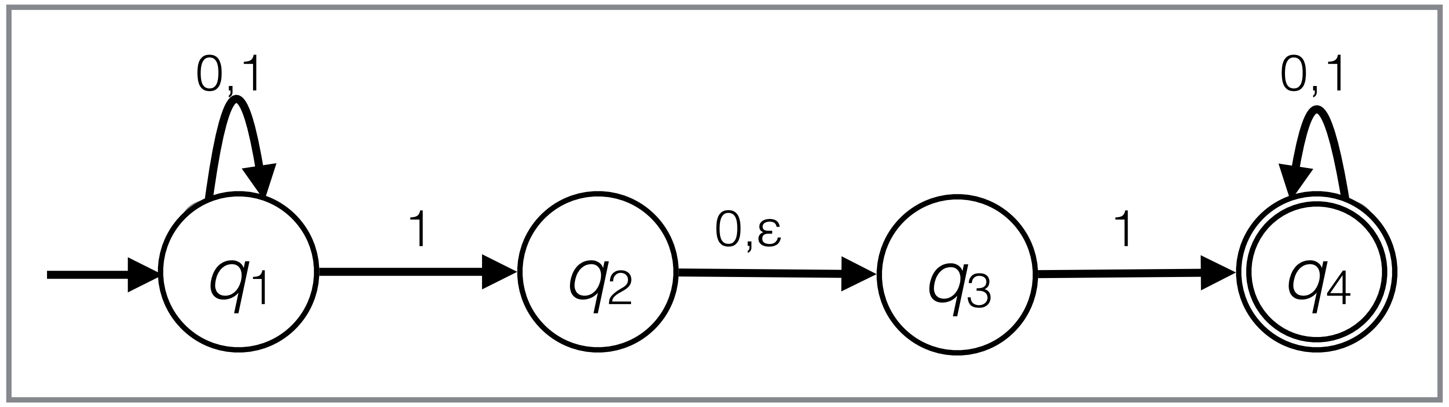
DEFINITION

A nondeterministic finite automaton (NFA) is a 5-tuple (Q,Σ,δ,q_0,F), where

1. Q is a finite set called the *states.*
2. Σ is a finite set called the *alphabet*.
3. δ: $Q \times \Sigma_\varepsilon \to \mathcal{P}Q$ is the *transition function.*
4. q_0 is the *start state*
5. $F \subseteq Q$ is the set of *accept states* (or *final states*)

Note: In 3, Σ_ε is an abbreviation of $\Sigma \cup \{\varepsilon\}$

Definition of an NFA - Example


$$\begin{tikzcd}
\arrow[r] & q_1 \arrow[loop above, "{0,1}"] \arrow[r, "1"] & q_2 \arrow[r, "{0,\varepsilon}"] & q_3 \arrow[r, "1"] & \boxed{q_4} \arrow[loop above, "{0,1}"]
\end{tikzcd}$$

The formal description of this NFA is (Q,Σ,δ,q_1,F), where:

1. $Q = \{q_1,q_2,q_3,q_4\}$, 2. $\Sigma = \{0,1\}$

3. δ is given by:

	0	1	ε
q_1	$\{q_1\}$	$\{q_1,q_2\}$	$\varnothing$
q_2	$\{q_3\}$	$\varnothing$	$\{q_3\}$
q_3	$\varnothing$	$\{q_4\}$	$\varnothing$
q_4	$\{q_4\}$	$\{q_4\}$	$\varnothing$

4. q_1 is the start state, and 5. $F = \{q_4\}$

Formally defining acceptance for NFAs

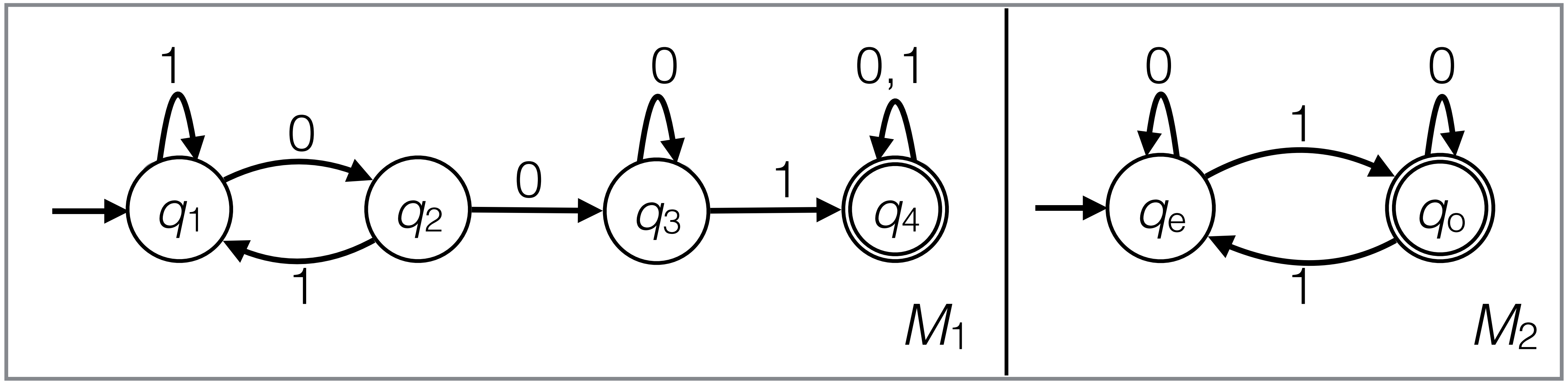
DEFINITION

Let $N = (Q,\Sigma,\delta,q_0,F)$ be an NFA, and w a string over Σ. Then N accepts w if we can write w as $w = y_1y_2\cdots y_m$, where each y_i is a member of Σ_ε and there is some sequence of states $r_0,r_1,\ldots,r_m$ in Q such that:

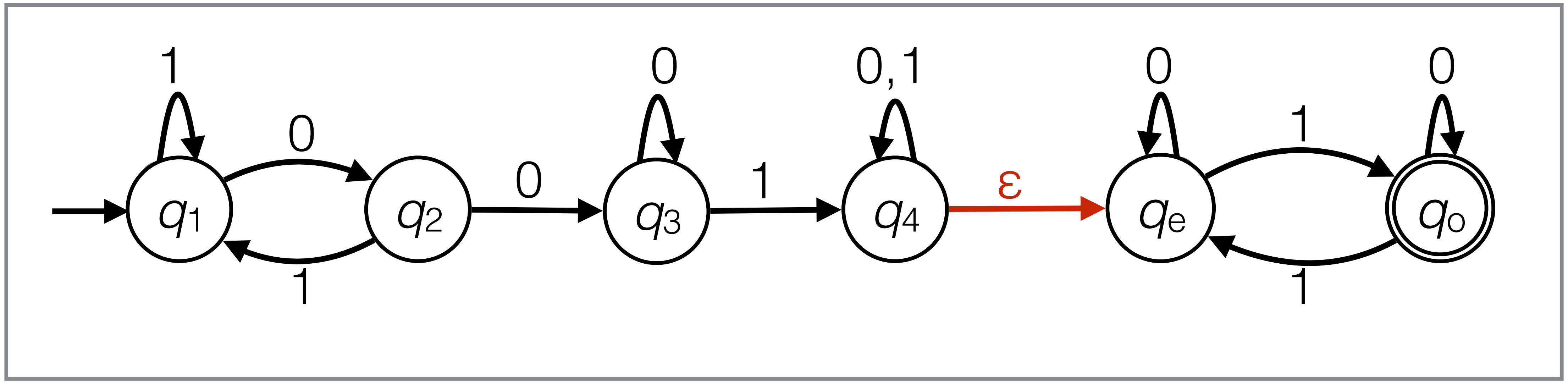
1. $r_0 = q_0$
2. $r_{i+1} \in \delta(r_i,y_{i+1})$ for $i = 0,1,\ldots,m\text{-}1$
3. $r_m \in F$

Combining DFAs into an NFA

Recall that we were trying to show that if *A,B* are regular then so is $A \circ B$. If *A,B* are recognised by M_1, M_2 respectively



then here is an **NFA** that recognises $A \circ B$:



Combining DFAs into an NFA

- *As we will confirm later*, this is possible in **all** cases, for **any** *A,B.* That is, if $L(M_1) = A$ and $L(M_2) = B$, then there is a way to construct an NFA N out of M_1, M_2 such that N recognises $A \circ B$ (and similarly for the other 2 regular operations union and star).

- How does this help us prove the theorem on slide 2? We wanted a **DFA** to recognise $A \circ B$.

- In fact it helps because it turns out that DFAs and NFAs **recognise the same languages**, i.e., for every NFA N there is a DFA M such that $L(N) = L(M)$.

CARDIFF
UNIVERSITY
PRIFYSGOL
CAERDYDD

NFAs and DFAs are Equivalent!

Equivalence of NFAs and DFAs

DEFINITION
We say that 2 machines are equivalent if they recognise the same language.

Then we have the following theorem:

THEOREM
Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Remark We already know the converse of the above theorem holds, i.e., every DFA has an equivalent NFA. How do we know?

Proving equivalence of NFAs and DFAs

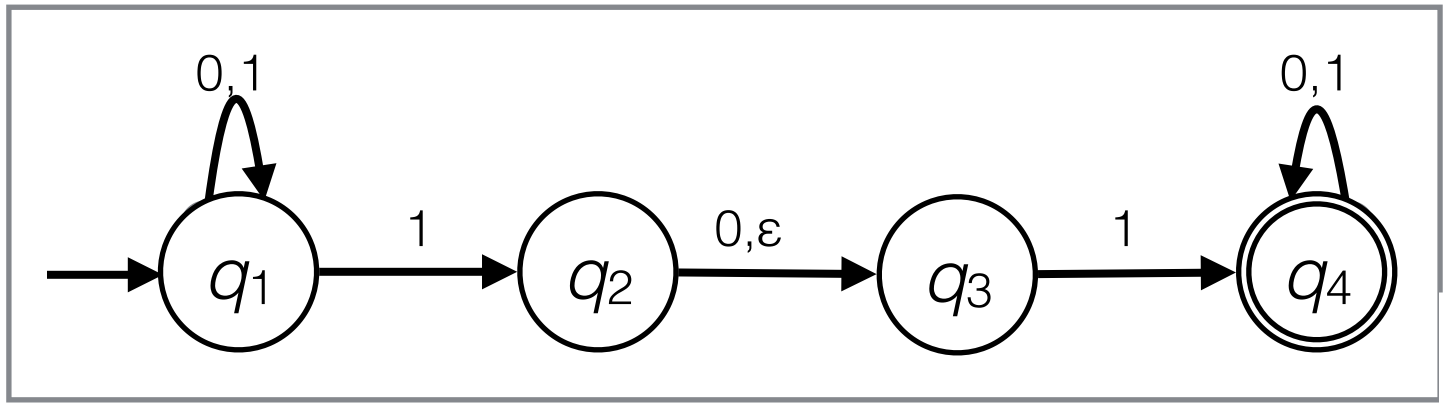
PROOF (outline)

Let N be a given NFA (Q,Σ,δ,q_0,F). We must construct a DFA $M = (Q',\Sigma,\delta',q_0',F')$ such that M recognises the same language as N. We look at 2 different cases:

1. *N* has no ε-arrows.
2. *N* does have ε-arrows.

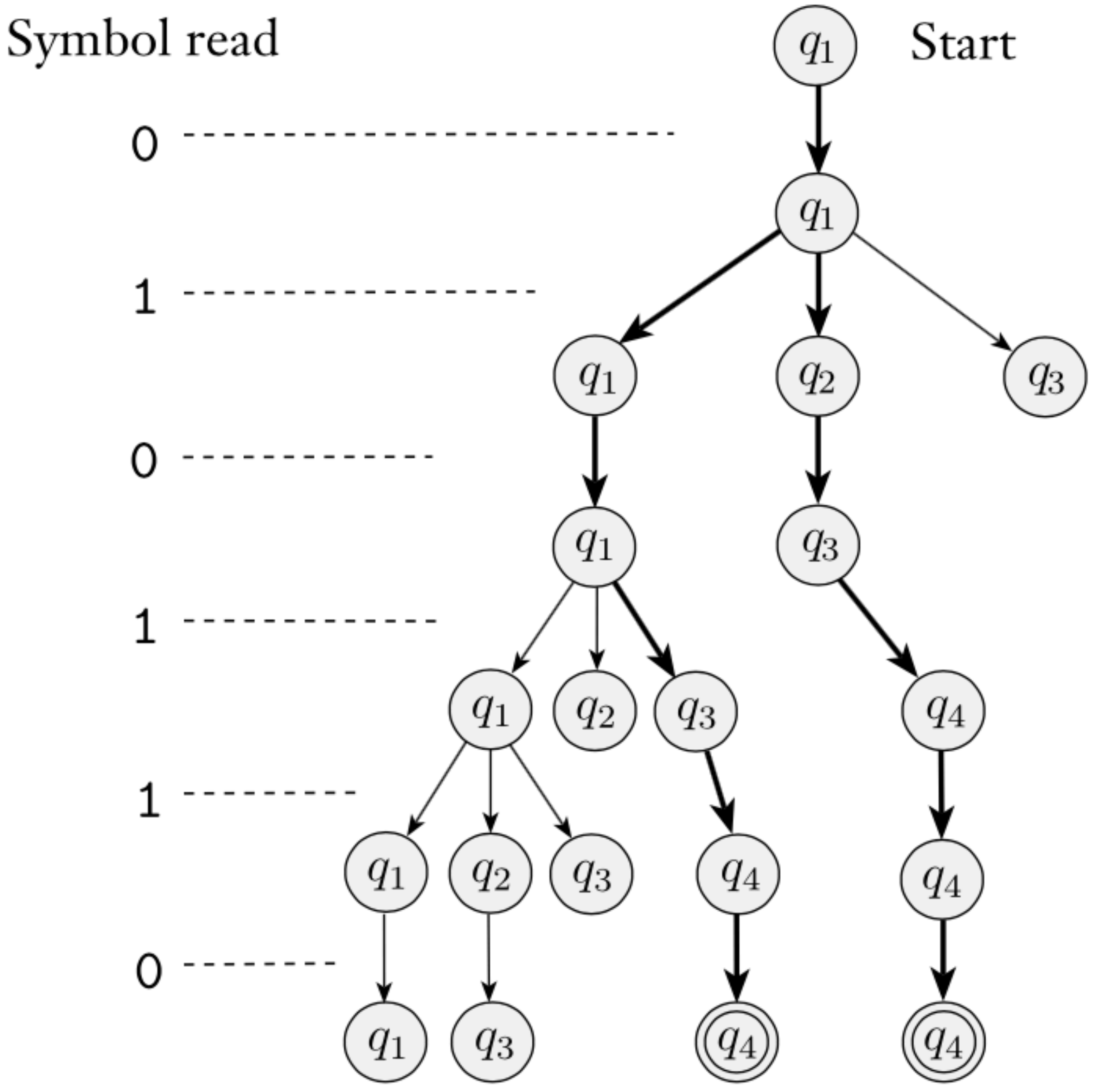
(Note: alphabet Σ will be the same for M and N)

Example



$$\begin{tikzcd}
{} \arrow[r] & q_1 \arrow[loop above, "{0,1}"] \arrow[r, "1"] & q_2 \arrow[r, "{0,\varepsilon}"] & q_3 \arrow[r, "1"] & q_4 \arrow[loop above, "{0,1}"]
\end{tikzcd}$$

Computation on input 010110



Can visualise the computation as a tree[†] $\longrightarrow$

At least one branch ends in an accept state (q_4) so string is accepted

† A tree is a connected graph with no simple cycles

Proving equivalence of NFAs and DFAs

PROOF (outline) (contd.)

Case 1: *N* has no ε-arrows.

We define each of $Q', \Sigma, \delta', q_0', F'$ in turn:

- $Q' = \mathcal{P}Q$ (i.e., every state of *M* is a *set* of states of *N*).
- For $R \in Q'$, $a \in \Sigma$, let

$$\delta'(R,a) = \{q \in Q \mid q \in \delta(r,a) \text{ for some } r \in R\}$$

- $q_0' = \{q_0\}$
- $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$

See “Example translation from NFA to DFA” below Video 4, Week 3 in Learning Central

Proving equivalence of NFAs and DFAs

PROOF (outline) (contd.)

Case 2: N has ε-arrows.

[Notation: for $R \in Q'$, $E(R) = \{q \mid q$ can be reached from a member of R by moving along 0 or more ε-arrows in $N\}$]

Q', F' as in case 1. Modify δ', q_0' as follows:

- $\delta'(R,a) = \{q \in Q \mid q \in E(\delta(r,a))$ for some $r \in R\}$
- $q_0' = E(\{q_0\})$

This completes the construction of M, which recognises the same language as N.

Regular languages and NFAs

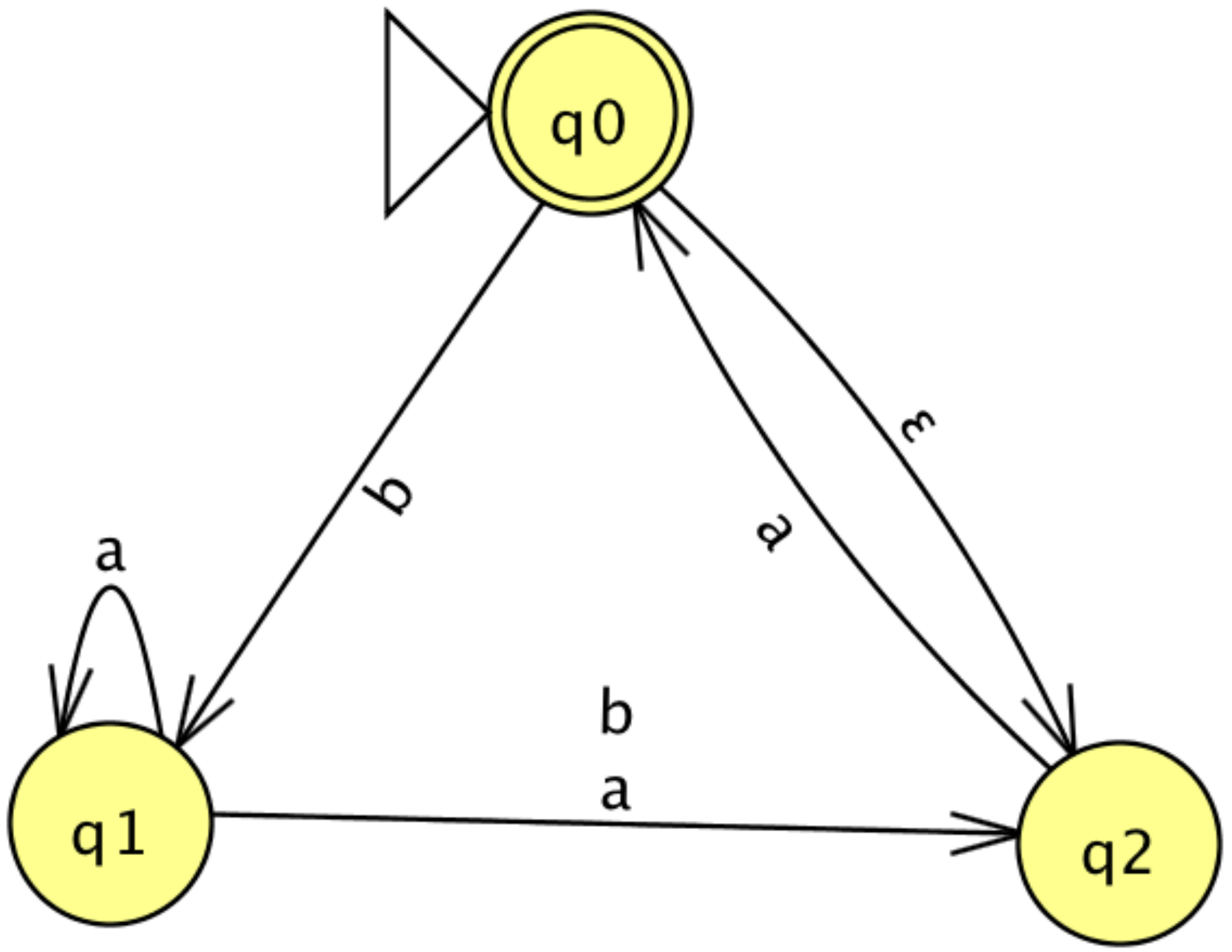
The previous theorem has the following corollary:

COROLLARY
A language is regular iff it is recognised by some NFA.

Properties of Regular Languages

Converting an NFA to a DFA : Example in JFLAP

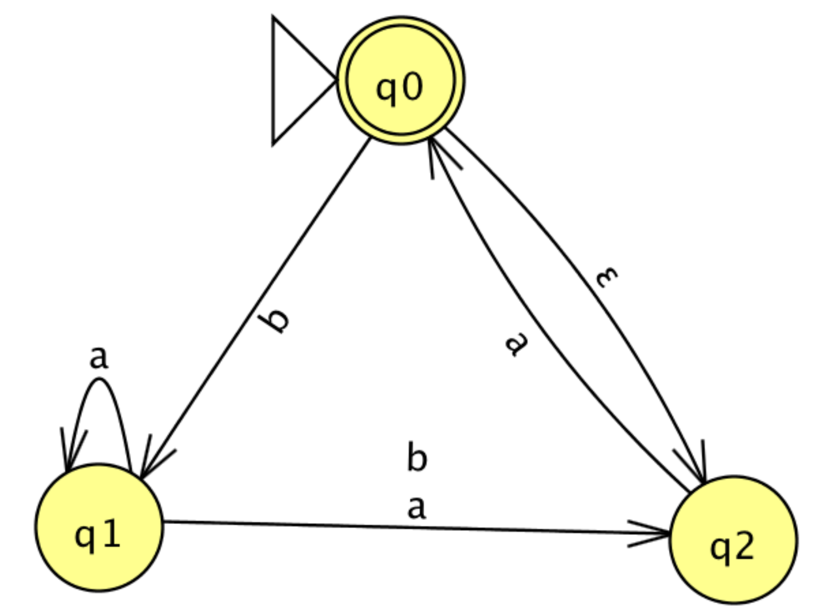
Consider the following NFA N_1:



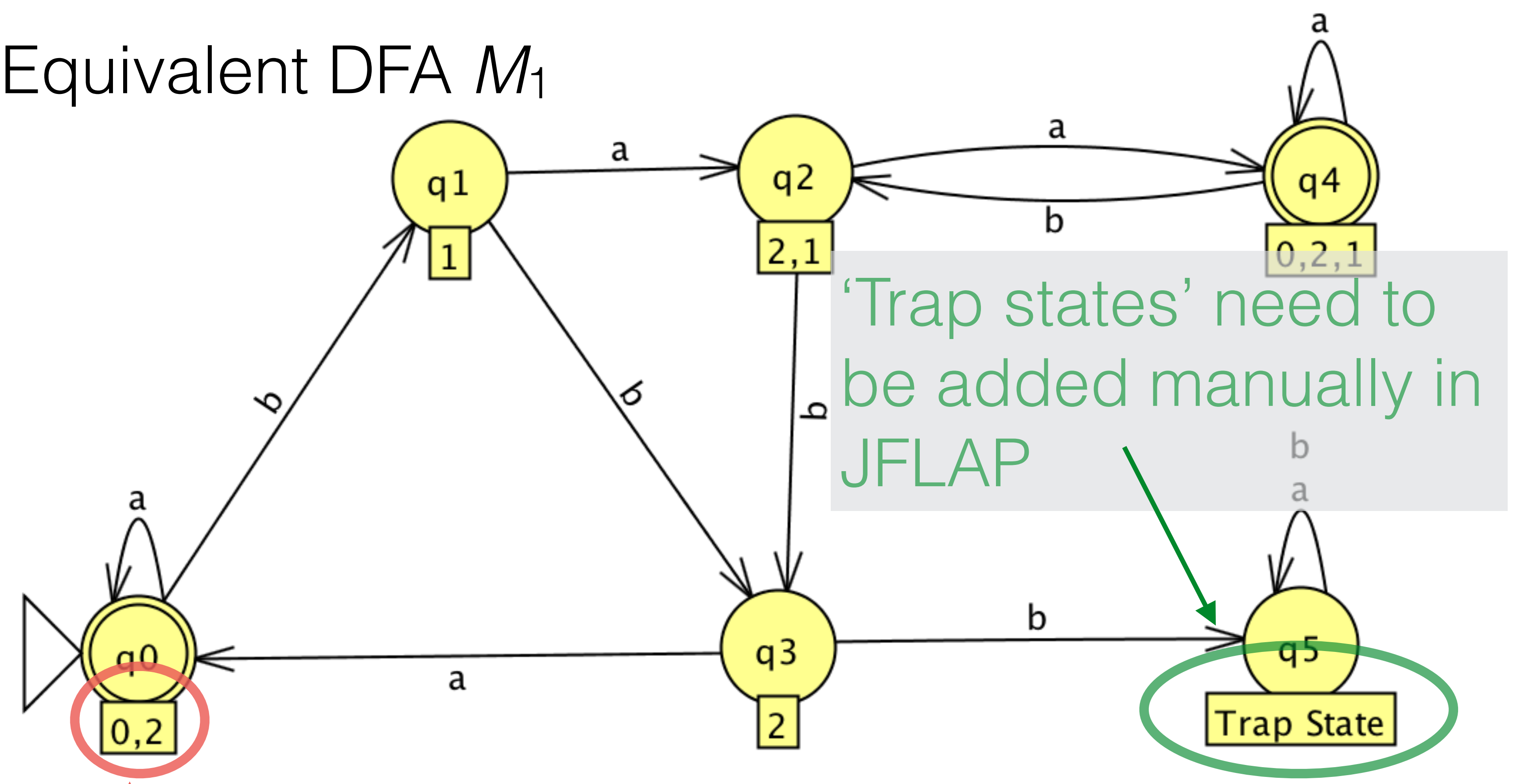
Converting an NFA to a DFA : Example in JFLAP

On the right is the DFA generated with JFLAP that is equivalent to N_1.

N_1



Equivalent DFA M_1



‘Trap states’ need to be added manually in JFLAP

This box gives the corresponding set of states from N_1, in this case $\{q_0, q_2\}$

Equivalence of NFAs and DFAs

Last time we provided a way to convert any NFA into a DFA that recognised the same language. Thus proving:

THEOREM
Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

This theorem has the following corollary:

COROLLARY
A language is regular iff it is recognised by some NFA.

Regular languages are closed under regular operations (finally!)

With this corollary in hand, we are now finally in a position to prove the following result, which we started in last week's Video 1

<u>THEOREM</u>

If A and B are regular languages, then so are $A \cup B$, $A \circ B$ and A^*.

Let's run through the proofs of each of these 3 in turn, starting with $A \circ B$.

Regular languages are closed under concatenation

PROOF

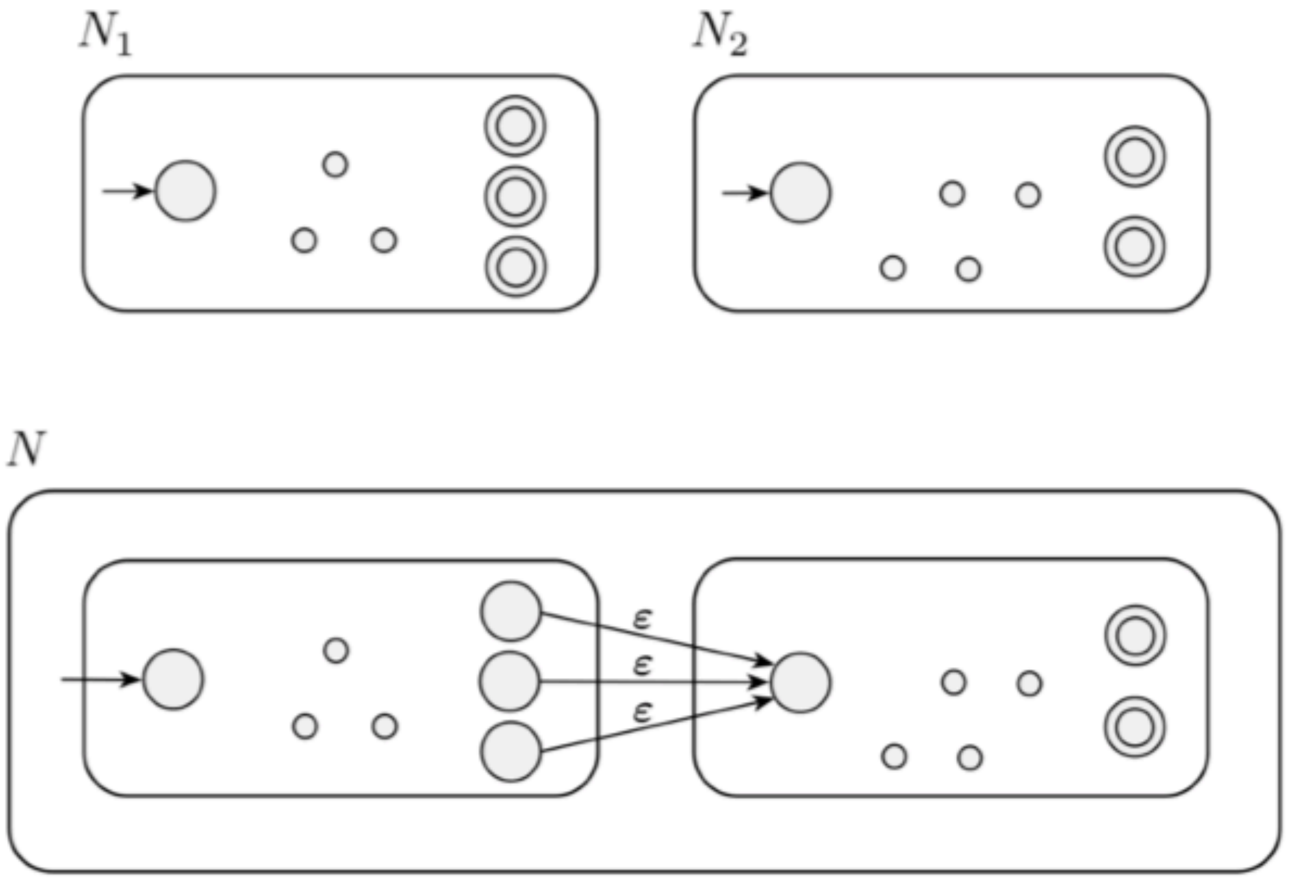
Let's assume A,B are regular.
We want to show $A \circ B$ is regular. By the previous corollary it suffices to show there exists NFA N such that $L(N) = A \circ B$.
Since A,B are regular we know there exist NFAs N_1, N_2 such that $L(N_1) = A$ and $L(N_2) = B$.

Let's construct N from N_1, N_2 according to the method outlined on the next slide…

Building an NFA to recognise $A \circ B$

Pictorially:

(Pics from Sipser book)



N nondeterministically "guesses" when to split the input into its initial part (accepted by N_1) and the 2nd part (by N_2).

Building an NFA to recognise $A \circ B$

Formally: Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Then $N = (Q, \Sigma, \delta, q_1, F_2)$, where:

1. $Q = Q_1 \cup Q_2$ (states of N are all states of N_1, N_2)
2. Start state q_1 of N is same as that of N_1.
3. Accept states F_2 are same as those of N_2.
4. δ is defined as follows, for any $(q,a) \in Q \times \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & \text{if } q \in F_1,\ a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & \text{if } q \in F_1,\ a = \varepsilon \\ \delta_2(q,a) & \text{if } q \in Q_2 \end{cases}$$

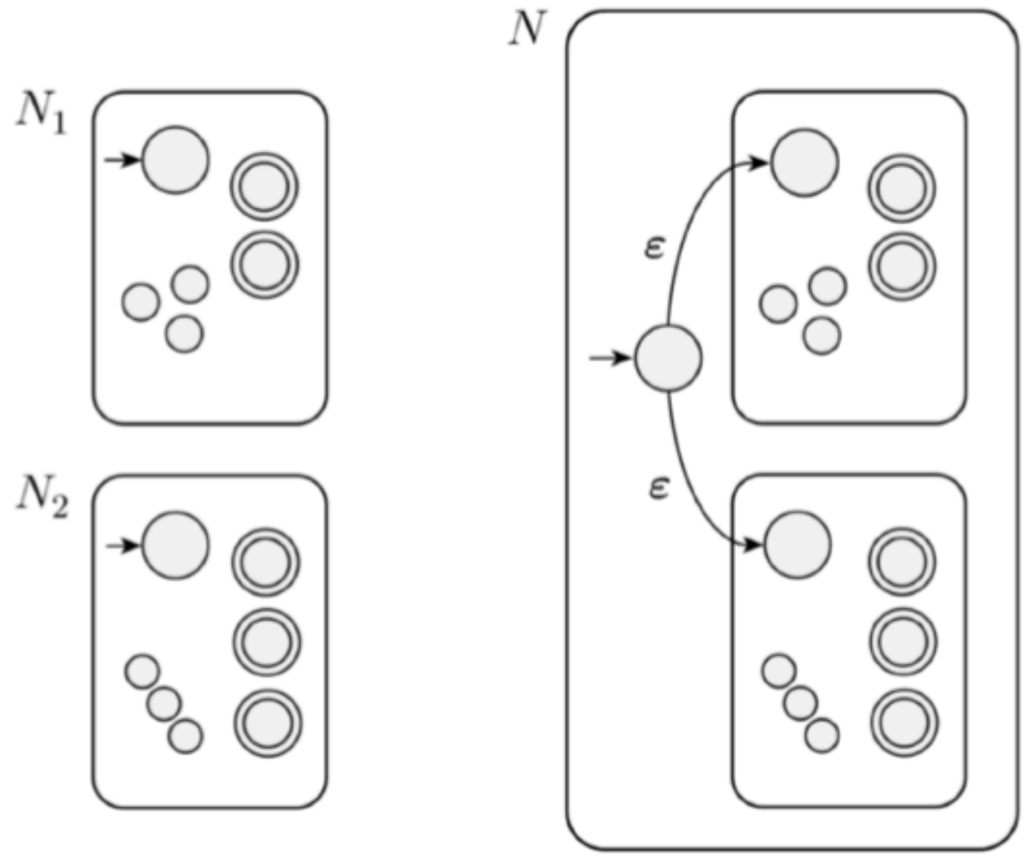
Building an NFA to recognise $A \circ B$

N recognises $A \circ B$. Hence, there is an NFA that recognises $A \circ B$ and so $A \circ B$ is regular by the previous corollary.

Next, to show $A \cup B$ is regular we construct a different NFA N from N_1, N_2 as on the next slide.

Building an NFA to recognise *A* ∪ *B*

Pictorially:



At start of input, *N* nondeterministically “guesses” which of the 2 machines N_1,N_2 accepts the input.

Building an NFA to recognise $A \cup B$

Formally: Let $N_1 = (Q_1,\Sigma,\delta_1,q_1,F_1)$, $N_2 = (Q_2,\Sigma,\delta_2,q_2,F_2)$.
Then $N = (Q,\Sigma,\delta,q_0,F)$, where:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$ (states of N are all states of N_1, N_2, plus a new start state q_0)

2. New state q_0 is the start state of N.

(continued on next slide)

Building an NFA to recognise $A \cup B$

Formally: Let $N_1 = (Q_1,\Sigma,\delta_1,q_1,F_1)$, $N_2 = (Q_2,\Sigma,\delta_2,q_2,F_2)$. Then $N = (Q,\Sigma,\delta,q_0,F)$, where:

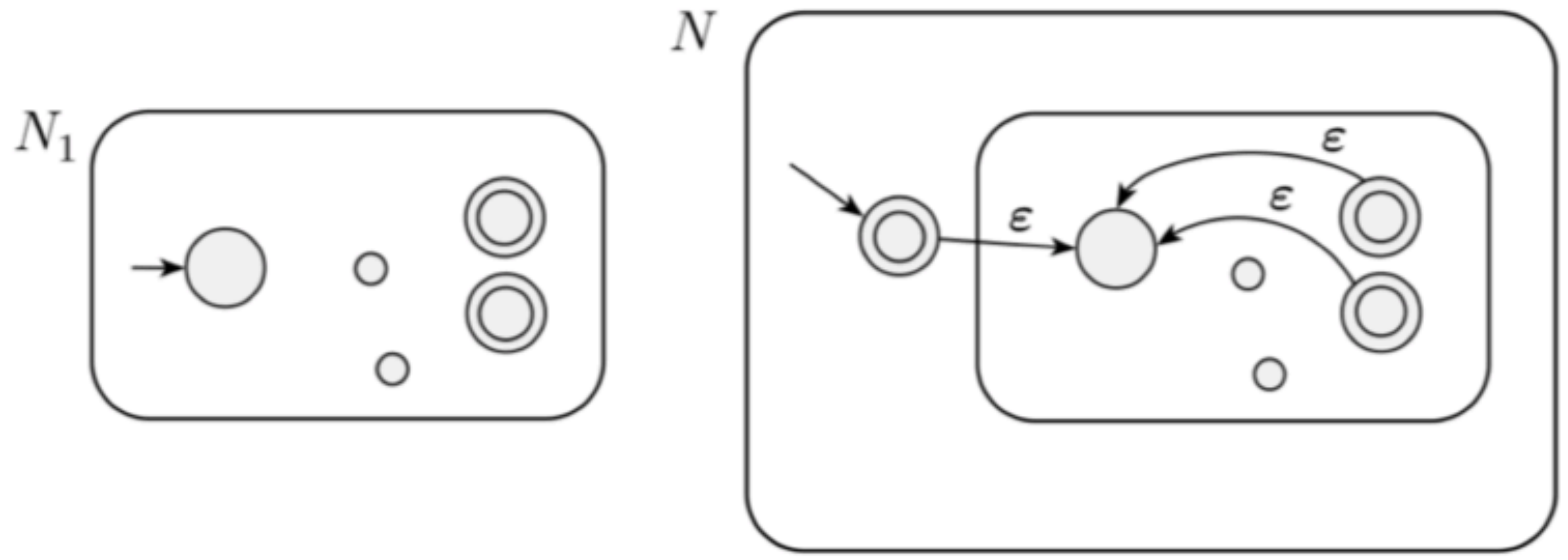
3. $F = F_1 \cup F_2$ (the accept states F of N are all the accept states of N_1,N_2.)

4. δ is defined as follows, for any $(q,a) \in Q \times \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1 \\ \delta_2(q,a) & \text{if } q \in Q_2 \\ \{q_1,q_2\} & \text{if } q = q_0,\ a = \varepsilon \\ \varnothing & \text{if } q = q_0,\ a \neq \varepsilon \end{cases}$$

Building an NFA to recognise *A**

Pictorially: (recall $A^* = \{x_1x_2\cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$)



- *N* has the option of "jumping back" to the start state (of N_1) to read another piece of input that N_1 accepts.
- We need a new start state to ensure ε is accepted (because always $\varepsilon \in A^*$ for **any** language *A*)

Building an NFA to recognise A^*

Formally: Let $N_1 = (Q_1,\Sigma,\delta_1,q_1,F_1)$ recognise A.
Then $N = (Q_1,\Sigma,\delta,q_0,F)$, where:

1. $Q = \{q_0\} \cup Q_1$ (states of N are all states of N_1 plus a new start state q_0)

2. New state q_0 is the start state of N.

(continued on next slide)

Building an NFA to recognise A^*

Formally: Let $N_1 = (Q_1,\Sigma,\delta_1,q_1,F_1)$ recognise A.
Then $N = (Q_1,\Sigma,\delta,q_0,F)$, where:

3. $F = \{q_0\} \cup F_1$ (the accept states are the old accept states plus the new start state.)

4. δ is defined as follows, for any $(q,a) \in Q \times \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & \text{if } q \in F_1,\ a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & \text{if } q \in F_1,\ a = \varepsilon \\ \{q_1\} & \text{if } q = q_0,\ a = \varepsilon \\ \varnothing & \text{if } q = q_0,\ a \neq \varepsilon \end{cases}$$

Regular Expressions

What have we achieved until now?

- We have 2 equivalent ways to describe regular languages:
    - They are the languages recognised by DFAs
    - They are the languages recognised by NFAs
- We've shown the class of regular languages is closed under union, concatenation, star.
- We now turn to a 3rd representation of regular languages, that doesn't refer to any kind of machine at all! Instead it is based on regular expressions.

Regular expressions

- Regular expressions provide a shorthand way to describe languages.

- Example: The regular expression

$$(0 \cup 1)0^*$$

  stands for the language consisting of all strings starting with 0 or 1 followed by any number of 0s and no 1s

- Regular expressions have an important role in computer science applications that involve text. Users may want to search for strings satisfying certain patterns specified using regular expressions.

Formal definition of a regular expression

DEFINITION

We say R is a regular expression if R is:

1. a, for some a in the alphabet Σ,
2. ε,
3. $\varnothing$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,
6. $(R_1{}^*)$, where R_1 is a regular expression.

Notes on the definition of regular expression

- In 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and {ε} respectively. In 3 the regular expression ∅ represents the empty language. In 4,5,6 the expressions represent the languages obtained by taking the union or concatenation of languages R_1, and R_2, or the star of R_1, respectively.

Notes on the definition of regular expression

- The definition of regular expression is another example of a *recursive* (or *inductive*) definition. That is, we have base cases 1,2,3 and then 4,5,6 tell how to build more complex cases out of simpler ones.

- Parentheses in an expression may be omitted. In this case the order of precedence is: star first, then concatenation, then union. Furthermore, "$R_1 \circ R_2$" is often written "R_1R_2".

Example: $(0\cup1)1^*$ is really $((0\cup1)\circ(1^*))$

and not, e.g., $((0\cup1)\circ1)^*$

A bit more notation

- R^+ is shorthand for RR^*. (R^+ represents the set of all strings made up of 1 or more concatenated strings from R)

- R^k is shorthand for the concatenation of k R's with each other, i.e., $RR\cdots R$ (k times).

- If $\Sigma = \{0,1\}$ we can write Σ as shorthand for the regular expression $0 \cup 1$. (So $\Sigma^* = (0 \cup 1)^*$ represents the set of all strings over Σ as before.)

Examples of regular expressions

Here are some examples of regular expressions and the languages they represent (assuming $\Sigma = \{0,1\}$)

- $0^*10^* = \{w \mid w \text{ contains a single '1'}\}$
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one '1'}\}$
- $(\Sigma\Sigma)^* = \{w \mid w \text{ has even length}\}$
- $\emptyset^* = \{\varepsilon\}$

Quick exercise

Let R be a regular expression. Are the following statements true or false?

- $R \circ \varepsilon$ and R represent the same language
  - **TRUE**
- $R \circ \emptyset$ and R represent the same language
  - **FALSE** (unless $R = \emptyset$)
- $R \cup \emptyset$ and R represent the same language
  - **TRUE**
- $R \cup \varepsilon$ and R represent the same language
  - **FALSE** (unless R represents a language that already contains the empty string)

Equivalence of regular expressions and DFAs/NFAs

What kinds of languages can be described by regular expressions? In fact exactly the same ones that can be recognised by DFAs/NFAs:

THEOREM (KLEENE'S THEOREM)
A language is regular iff some regular expression describes it.

Kleene’s Theorem

Equivalence of regular expressions and DFAs/NFAs

- The result which establishes the equivalence of regular expressions with DFAs/NFAs is Kleene's Theorem:

<u>THEOREM</u> (KLEENE'S THEOREM)
A language is regular iff some regular expression describes it.

- Let's try to prove this result!

Proving Kleene's Theorem

First notice this result claims an "iff" ("if, and only if") (also sometimes written as "$\Leftrightarrow$") statement.

Such statements can be broken down into 2 "halves" like follows:

1. "if" part, or "$\Leftarrow$" direction

   A language is regular if some regular expression describes it.

2. "only if" part, or "$\Rightarrow$" direction

   A language is regular only if some regular expression describes it.

Proving Kleene's Theorem

We can prove an "iff" statement by proving each of these 2 halves in turn. Let's prove Kleene's Theorem by first proving its "if" part.

1. "if" part, or "⇐" direction
   A language is regular if some regular expression describes it.

Proving Kleene's Theorem - the "if" part

PROOF

Let *A* be a language that is described by a regular expression *R*. We must show *A* is a regular language. By our previous results, one way to do this is to construct (from *R*) an NFA that recognises *A*. What this NFA will look like depends on the form of *R.*
According to the definition of regular expression, *R* can take one of 6 forms.
We consider the base cases 1-3 first.

Formal definition of a regular expression

DEFINITION

We say R is a regular expression if R is:

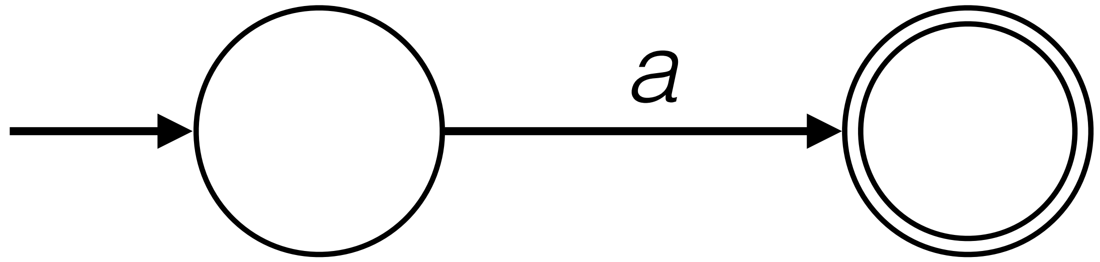
1. a, for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,
6. $(R_1{}^*)$, where R_1 is a regular expression.

Proving Kleene's Theorem - the "if" part

PROOF (Contd)

1. *R* = *a*, for some *a* in the alphabet ∑

   Then *R* describes the language {*a*}, which is recognised by the following NFA.

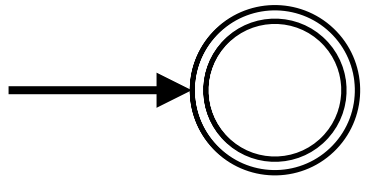


Proving Kleene's Theorem - the "if" part

PROOF (Contd)

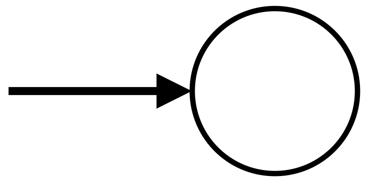
2. *R* = ε

   Then *R* describes the language {ε}, which is recognised by the following NFA.



3. *R* = ∅

   Then *R* describes the language ∅, which is recognised by the following NFA.



Proving Kleene's Theorem - the "if" part

PROOF (Contd)

Next we consider the inductive cases 4-6 of the definition of regular expression. In each case we show the result holds for the complex expression (e.g.,$R_1 \cup R_2$) when we assume it already holds for the simple expressions (R_1,R_2) from which it is composed.

[This is essentially another example of *proof by induction*, but with induction being performed over the *structure* of *R*.]

Proving Kleene's Theorem - the "if" part

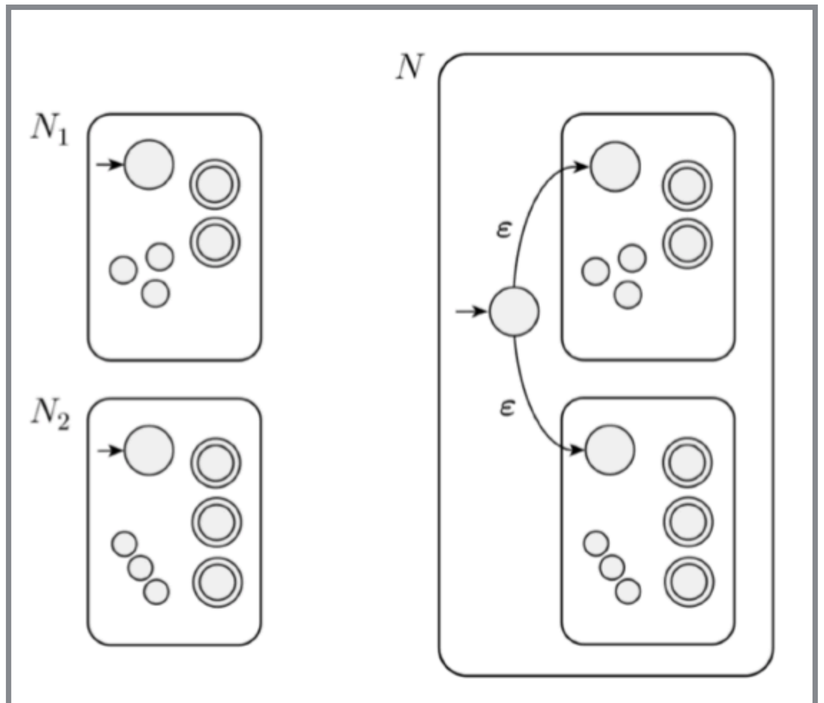
PROOF (Contd)

So, we must show that, assuming the languages described by R_1 and R_2 are recognised by NFAs N_1 and N_2 resp., then there exist NFAs that recognise each of the languages described by $R_1 \cup R_2$, $R_1 \circ R_2$ and R^*.

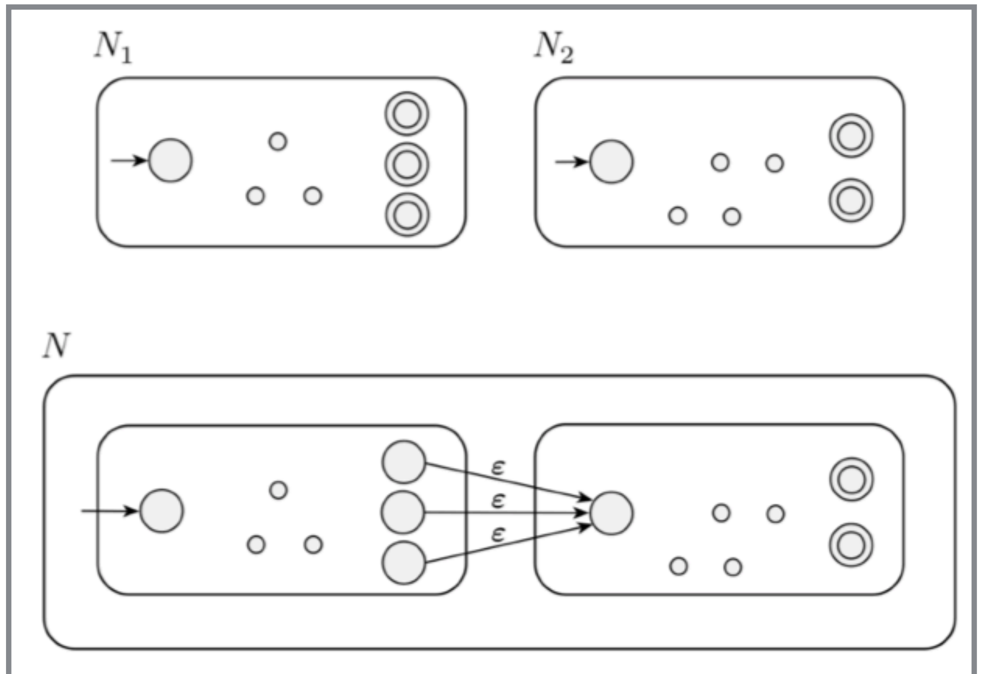
We can just re-use the constructions from our proof on the closure of regular languages under union, concatenation and star.

Proving Kleene's Theorem - the "if" part

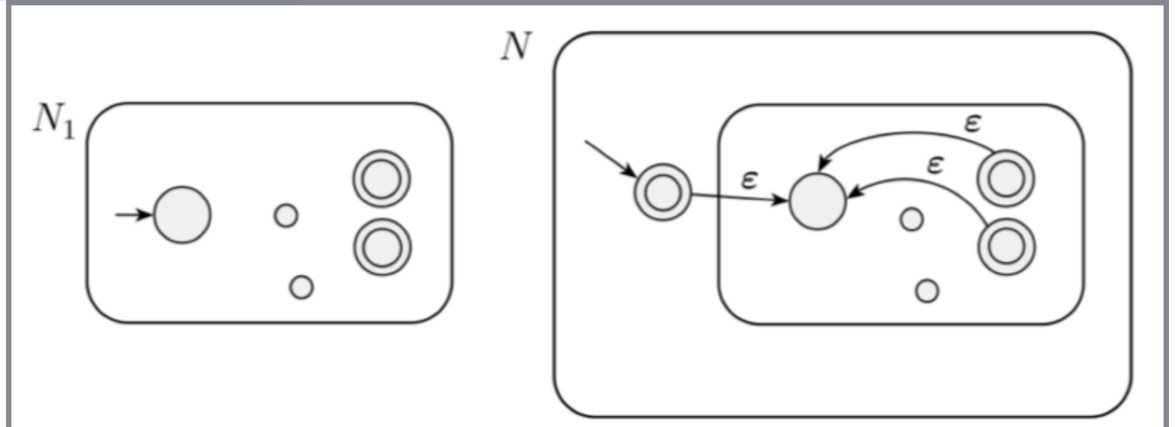
PROOF (Contd)



$R_1 \cup R_2$



$R_1 \circ R_2$



$R_1{*}$

Building an NFA from a regular expression: An example

- This concludes the proof of the "if" part of Kleene's Theorem.

- Let's illustrate the construction method of the proof on a specific example.

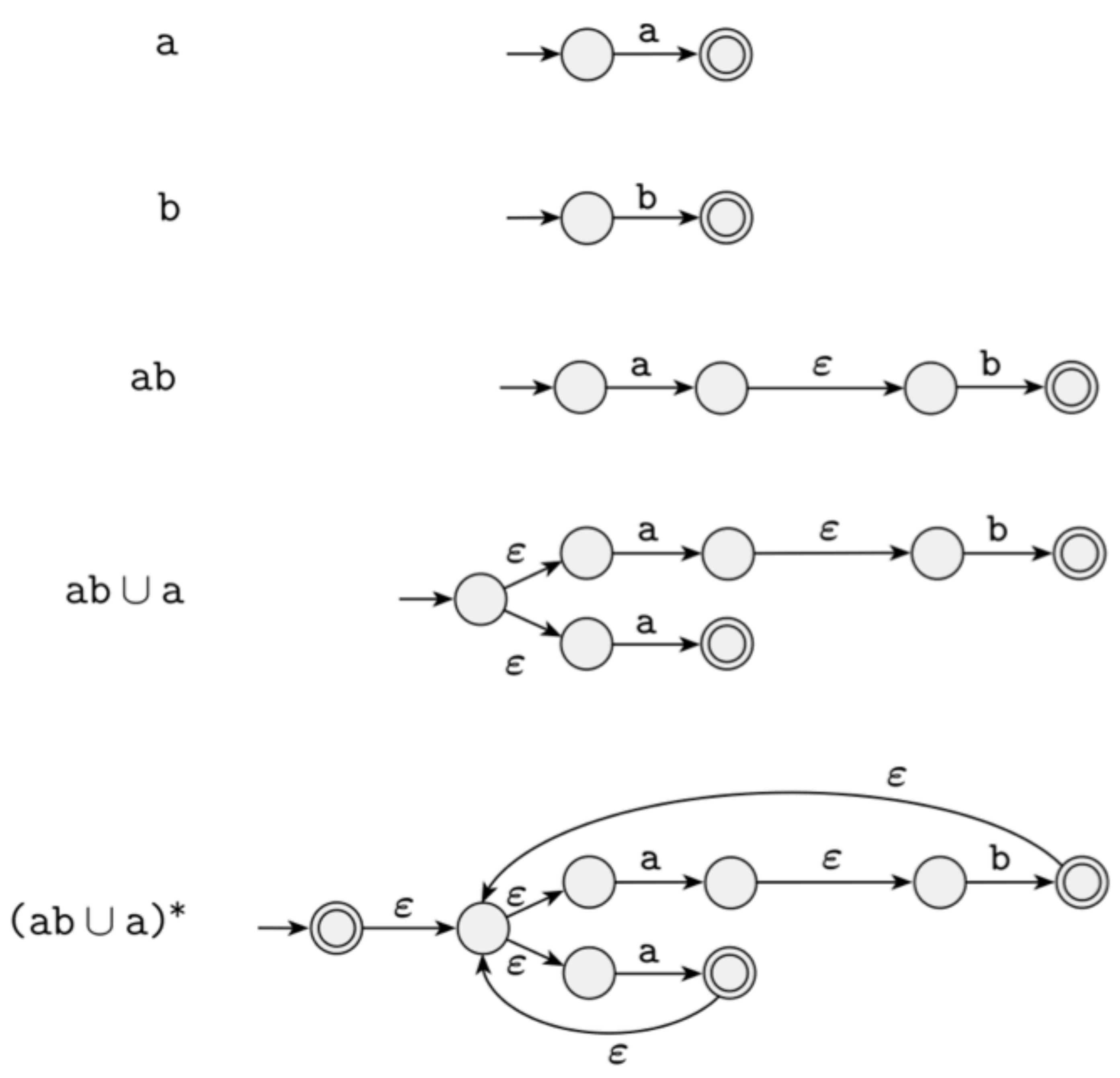
Example: Consider the regular expression (abUa)*

(Question: What is the language described by this expression?)

ANSWER: the set of strings in which every occurence of b is preceded by an occurrence of a

Building an NFA from a regular expression: An example

We can build up an NFA to recognise the described language (ab∪a)* step-by-step as follows:



Proving Kleene's Theorem - the "only if" part

So far we've only proved one half of Kleene's Theorem, i.e., the "if" part. What about the "only if" part.

2. "only if" part, or "⇒" direction

   A language is regular only if some regular expression describes it.

PROOF

Rather more involved than the "if" part! See Sipser book pages 69-76 if interested.

Non-regular Languages: The Pumping Lemma

Non-regular languages

- Up to now we've focussed on languages that **are** regular, on how they can be represented, and on building more complex ones out of simpler ones.

- What about the languages that **aren't** regular? Does one even exist?

- In fact the following language is a simple language that is **not** regular:

$$\{0^n1^n \mid n \geq 0\}$$

In particular there is **no** FA that can recognise this language.

- We can prove non-regularity of a language using a result known as the Pumping Lemma.

The Pumping Lemma

- The Pumping Lemma states that **all** regular languages have some particular property (specified in full next slide).

- Hence whenever we can prove a given language does **not** have this property, we can be sure that language cannot be regular.

- *Roughly*, the property is that each string longer than a certain length (the pumping length) contains a section that can be repeated any number of times with the resulting string remaining in the language.

The Pumping Lemma - formally

<u>THEOREM</u> (THE PUMPING LEMMA)
If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into 3 pieces $s = xyz$, satisfying the following conditions:

1. For each $i \geq 0$, $xy^iz \in A$.
2. $|y| > 0$, and
3. $|xy| \leq p$

The Pumping Lemma - illustration

Let *A* be a regular language. Then by the Pumping Lemma, it has some pumping length *p*. Let's assume $p = 5$.
Suppose the following string is in *A*: $s = 10110111$
We have $|s| = 8 > p$, so the Pumping Lemma says we can divide *s* into 3 pieces satisfying the 3 conditions. For example it might look like this:

$$\overset{x}{10} \,\vdots\, \overset{y}{11} \,\vdots\, \overset{z}{0111}$$

(note $|y| > 0$ and $|xy| < p$, so conditions 2,3 are satisfied here)
By condition 1 the following are all in *A* too:

$$\overset{x}{10} \,\vdots\, \overset{z}{0111} \qquad \overset{x}{10} \,\vdots\, \overset{y^2}{1111} \,\vdots\, \overset{z}{0111} \qquad \overset{x}{10} \,\vdots\, \overset{y^3}{111111} \,\vdots\, \overset{z}{0111}\text{, etc}$$

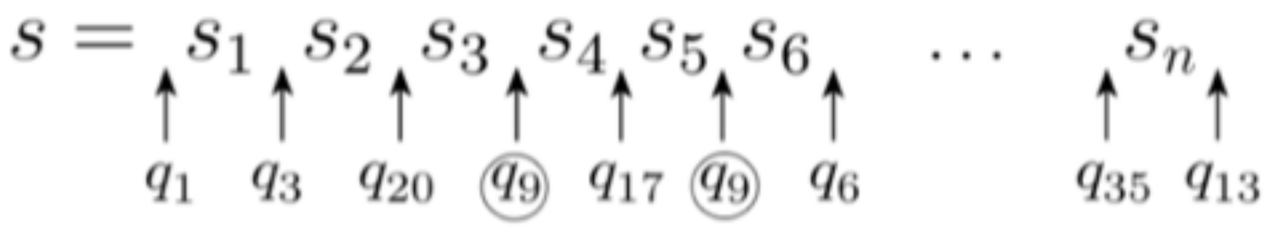
Proving the Pumping Lemma

PROOF (Outline)
Let A be a regular language. We need to show there is some p (the pumping length) that conforms with the behaviour stated in the Pumping Lemma. Since A is regular, we know from our previous results that there is some DFA $M = (Q,\Sigma,\delta,q_0,F)$ that recognises A.

Question: How can we determine p from this DFA M?

Proving the Pumping Lemma

PROOF (Outline) (Contd)
Let p = the number of states in Q. Then we must show any string s in A of length $\geq p$ can be broken into xyz satisfying the 3 conditions. Let s be in A with $|s| = n \geq p$. Consider the sequence of states M goes through when processing s. For example it might look something like this:

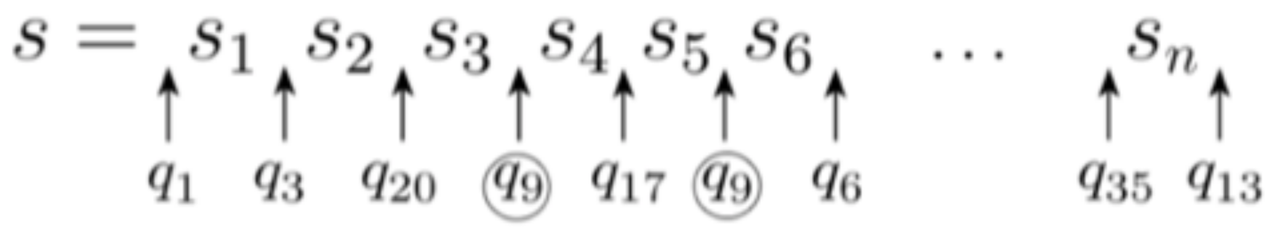


$s = s_1 \; s_2 \; s_3 \; s_4 \; s_5 \; s_6 \; \ldots \; s_n$
q_1
q_3
q_{20}
q_9
q_{17}
q_9
q_6
q_{35}
q_{13}

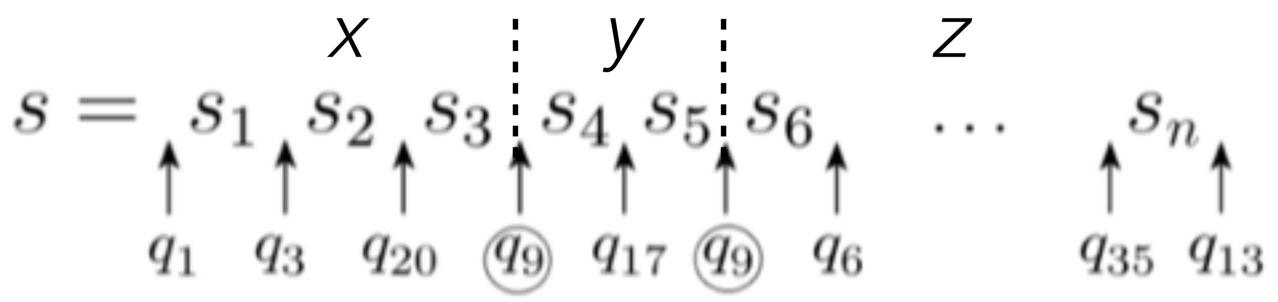
Note there will be n+1 states visited (including start state)

Proving the Pumping Lemma

PROOF (Outline) (Contd)


$s = s_1 s_2 s_3 s_4 s_5 s_6 \dots s_n$
q_1 q_3 q_{20} q_9 q_{17} q_9 q_6 q_{35} q_{13}

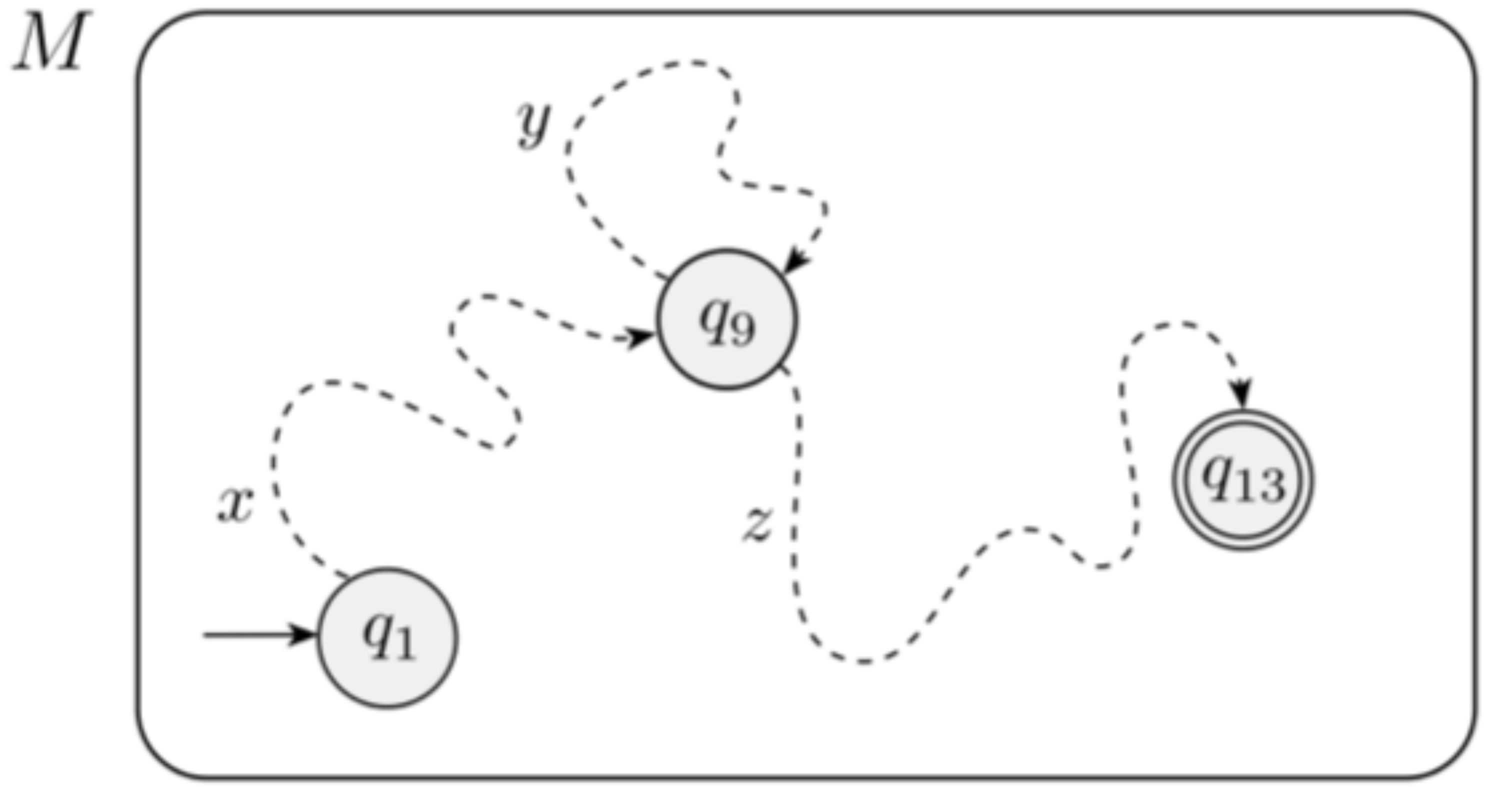
Since $n \geq p$ we know $n+1 > p$, so at least one state must be visited more than once. Take the first state in the sequence that appears more than once, for example let's assume it's q_9 above. Then we choose the following division of s.


x
y
z
$s = s_1 s_2 s_3 s_4 s_5 s_6 \dots s_n$
q_1 q_3 q_{20} q_9 q_{17} q_9 q_6 q_{35} q_{13}

i.e., y is between the 1st and 2nd occurrences of q_9

Proving the Pumping Lemma

<u>**PROOF**</u> (Outline) (Contd)



Then *xyz* satisfied the 3 conditions in the Pumping Lemma, as required.

Proving non-regularity of a language

- Let's now use the Pumping Lemma to prove the language

$$B = \{0^n 1^n \mid n \geq 0\}$$

  is **not** regular.

- Earlier we met the useful proof strategy "*proof by induction*". Now we employ another common strategy: "*proof by contradiction*"

PROOFS BY CONTRADICTION
To show a result is true, first assume it's false, and then show this assumption leads to an obviously false consequence, called a contradiction.

Proving non-regularity of a language

THEOREM

The language $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.

PROOF

Assume for contradiction B is regular. Then let p be the pumping length for B. Let $s = 0^p 1^p$. Since $s \in B$ and $|s| \geq p$ we know from Pumping Lemma that $s = xyz$ where, for any $i \geq 0$, $xy^i z \in B$. We consider 3 cases to show this is impossible.

(Continued next slide)

Proving non-regularity of a language

Case 1: *y* consists only of 0s. Then *xyyz* has more 0s than 1s so *xyyz* ∉ *B*, violating condition 1 from the Pumping Lemma - contradiction.

Case 2: *y* consists only of 1s. Same reasoning as above - contradiction.

Case 3: *y* contains both 0s and 1s. Then *xyyz* will have both some 0s appearing before 1s as well as the other way around, so *xyyz* ∉ *B* - contradiction.

Hence in all possible cases we obtain contradictions. Hence *B* is not regular, as required.

Context-free Grammars

Context-free grammars

- Recall that regular languages can be described via *regular expressions* such as (a∪b)*c etc.

- In the same way, context-free languages can be described using context-free grammars (CFGs).

- The hallmark of CFGs is that they are useful for describing features that have a *recursive* structure.

- This extends also to human languages, where sentences are formed by combining noun phrases and verb phrases, but where noun phrases may appear in verb phrases and vice versa.

Applications of CFGs

- In computer science, CFGs are applied to the specification and compilation of programming languages.

- Designers of compilers and interpreters often start by obtaining a grammar for that language.

- CFGs can be used to construct a parser that extracts the meaning of a program before generating the compiled code.

Starting with CFGs - informal example

The following is an example of a CFG G_1:

$$
\begin{aligned}
A &\rightarrow 0A1 \\
A &\rightarrow B \\
B &\rightarrow \#
\end{aligned}
$$

- G_1 has 3 substitution rules (or productions)
- The symbols to the left of the rules, i.e., A,B, are the variables.
- The variable of the first rule, i.e., A, is the start variable. All other symbols, i.e., 0,1,# are the terminals.

Building strings with a CFG

The substitution rules provide us with instructions for generating strings, by applying the following steps.

1. Write down the start variable.
2. Find a variable in the string currently written down and a rule that starts with that variable. Replace the variable with the right-hand side of that rule.
3. Repeat until no variable (i.e., only terminals) remain.

Starting with CFGs - informal example

Example G_1 continued:

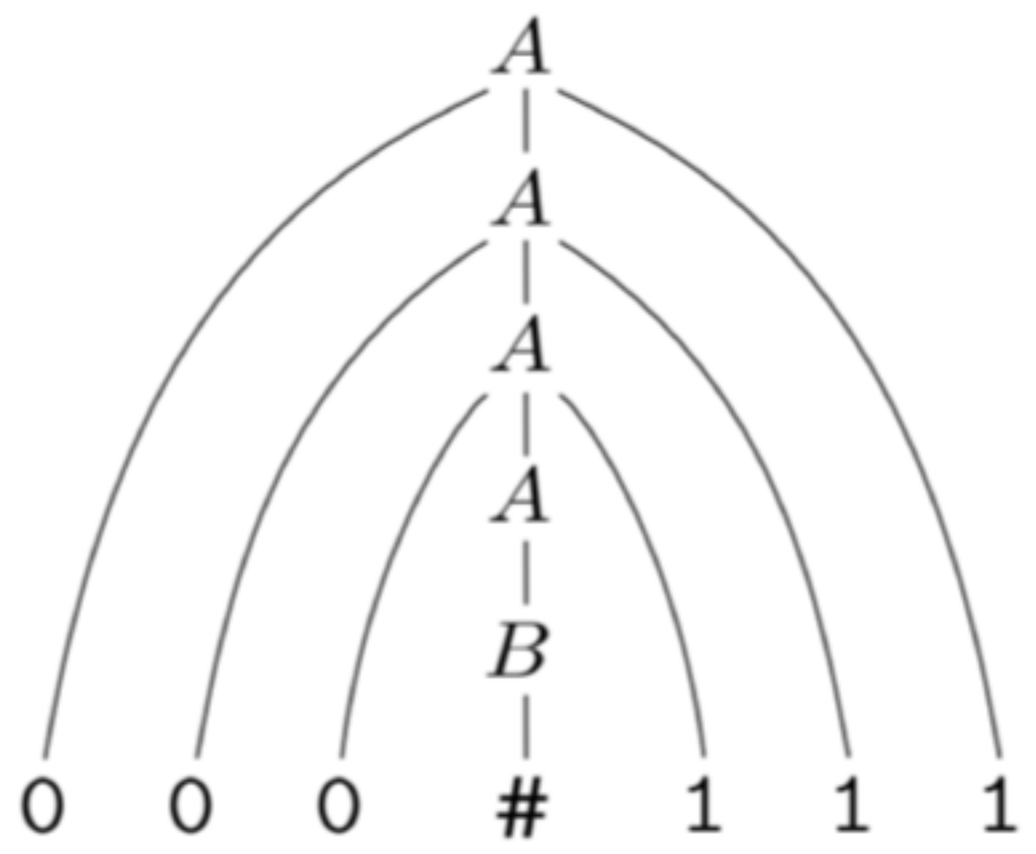
$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

G_1 generates string 000#111 via the following derivation:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Parse trees

Another way to visualise derivations is through a parse tree. This is a parse tree for the previous derivation.



Starting with CFGs - informal example

Example G_1 continued:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

G_1 also generates string # via the following derivation:

$$A \Rightarrow B \Rightarrow \#$$

Language of a grammar

- The set of all strings that can be generated from a given grammar G like this is called the language of that grammar, denoted L(G).
- Question: What is L(G_1), where G_1 is defined as on previous slides? **ANSWER:** L(G_1) = $\{0^n\#1^n \mid n \geq 0\}$
- Then a context-free language is any language that can be generated by some CFG.

Another example

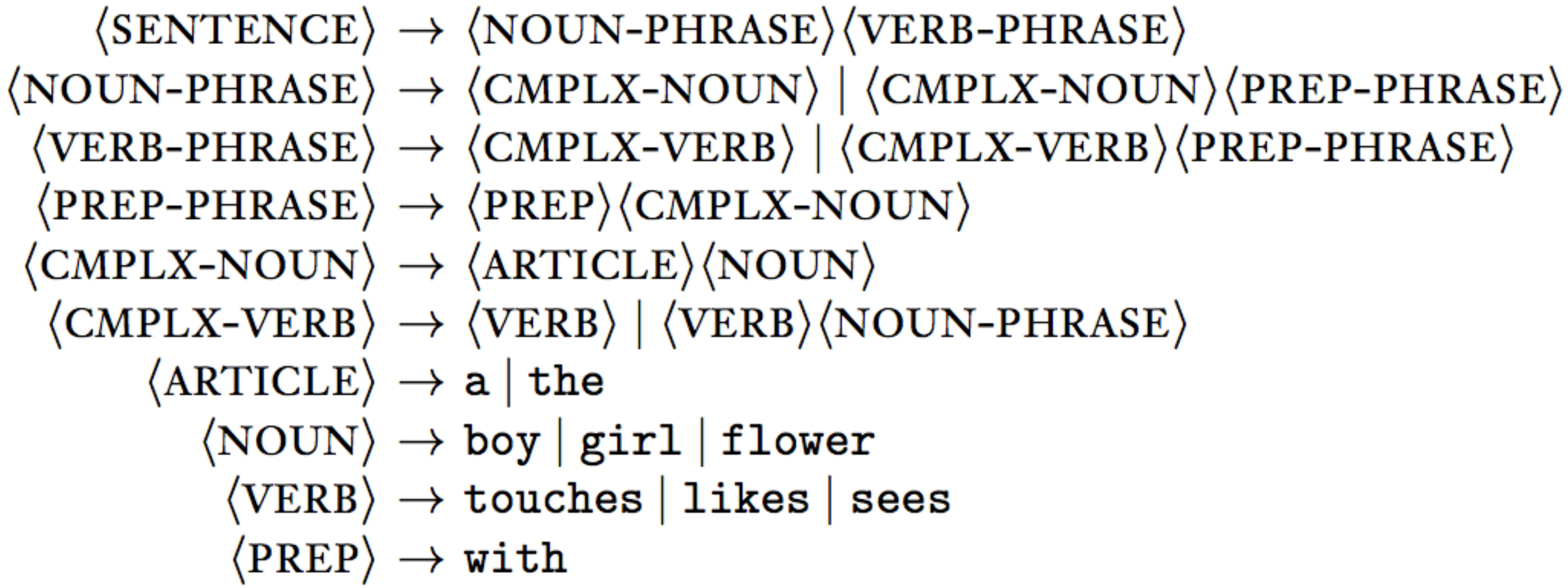
Consider the following grammar G_2:

$$\begin{aligned}
\langle\text{SENTENCE}\rangle &\to \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle \\
\langle\text{NOUN-PHRASE}\rangle &\to \langle\text{CMPLX-NOUN}\rangle \mid \langle\text{CMPLX-NOUN}\rangle\langle\text{PREP-PHRASE}\rangle \\
\langle\text{VERB-PHRASE}\rangle &\to \langle\text{CMPLX-VERB}\rangle \mid \langle\text{CMPLX-VERB}\rangle\langle\text{PREP-PHRASE}\rangle \\
\langle\text{PREP-PHRASE}\rangle &\to \langle\text{PREP}\rangle\langle\text{CMPLX-NOUN}\rangle \\
\langle\text{CMPLX-NOUN}\rangle &\to \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle \\
\langle\text{CMPLX-VERB}\rangle &\to \langle\text{VERB}\rangle \mid \langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle \\
\langle\text{ARTICLE}\rangle &\to \texttt{a} \mid \texttt{the} \\
\langle\text{NOUN}\rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle\text{VERB}\rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle\text{PREP}\rangle &\to \texttt{with}
\end{aligned}$$

Note: if we have several rules with the same variable, e.g., ⟨ARTICLE⟩ → a, ⟨ARTICLE⟩ → the, then we abbreviate by ⟨ARTICLE⟩ → a | the

Another example

Consider the following grammar *G_2*:



$$
\begin{aligned}
\langle\text{SENTENCE}\rangle &\to \langle\text{NOUN-PHRASE}\rangle\langle\text{VERB-PHRASE}\rangle\\
\langle\text{NOUN-PHRASE}\rangle &\to \langle\text{CMPLX-NOUN}\rangle \mid \langle\text{CMPLX-NOUN}\rangle\langle\text{PREP-PHRASE}\rangle\\
\langle\text{VERB-PHRASE}\rangle &\to \langle\text{CMPLX-VERB}\rangle \mid \langle\text{CMPLX-VERB}\rangle\langle\text{PREP-PHRASE}\rangle\\
\langle\text{PREP-PHRASE}\rangle &\to \langle\text{PREP}\rangle\langle\text{CMPLX-NOUN}\rangle\\
\langle\text{CMPLX-NOUN}\rangle &\to \langle\text{ARTICLE}\rangle\langle\text{NOUN}\rangle\\
\langle\text{CMPLX-VERB}\rangle &\to \langle\text{VERB}\rangle \mid \langle\text{VERB}\rangle\langle\text{NOUN-PHRASE}\rangle\\
\langle\text{ARTICLE}\rangle &\to \mathtt{a} \mid \mathtt{the}\\
\langle\text{NOUN}\rangle &\to \mathtt{boy} \mid \mathtt{girl} \mid \mathtt{flower}\\
\langle\text{VERB}\rangle &\to \mathtt{touches} \mid \mathtt{likes} \mid \mathtt{sees}\\
\langle\text{PREP}\rangle &\to \mathtt{with}
\end{aligned}
$$

Questions: How many rules, variables and terminals does *G_2* have? What is the start variable of *G_2*?

ANSWERS: 18 rules, 10 variables, 18 terminals (17 letters plus “space”). Start variable is 〈SENTENCE〉

Derivation exercise

Consider the following grammar *G*2:

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\to \texttt{with}
\end{aligned}
$$

Exercise: Give a derivation in *G*2 for the string a boy sees

ANSWER: See "**derivation of a boy sees**" under the video 1 in "Week 5 videos and slides" on LC.

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

Relation between Context-free and Regular languages

Formal definition of CFGs

DEFINITION

A context-free grammar is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the variables.
2. Σ is a finite set (with $\Sigma \cap V = \emptyset$), called the terminals.
3. R is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Formal definition of a derivation

- If u,v,w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say uAv yields uwv.

- We say u derives v, written $u \Rightarrow^* v$, if $u = v$ or if there exists a sequence $u_1,u_2,\ldots,u_k$ (for $k \geq 0$) and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

- The language of the grammar is $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$

Another example

Consider CFG $G_3 = (\{S\}, \{a,b\}, R, S)$, where R is given by:

$$S \rightarrow aSb \mid SS \mid \varepsilon$$

(note the empty-string symbol ε can appear on the right-hand side of a rule - but it is **not** considered to be a terminal.)

The following is a derivation of aabb:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

Relation between CFLs and regular languages

- There exist languages that **are** CFLs, but which are **not** regular.
- In fact one such language is $\{0^n1^n \mid n \geq 0\}$, which we already showed to be non-regular by the Pumping Lemma last lecture.

EXERCISE Give a CFG that generates $\{0^n1^n \mid n \geq 0\}$

ANSWER: $(\{S\}, \{0,1\}, R, S)$, with R given by

$$S \rightarrow 0S1 \mid \varepsilon$$

Relation between CFLs and regular languages

- We said that every regular language is also a CFL.
- This can be shown by proving that every regular language can be generated from some context-free grammar.
- In fact the regular languages are **exactly** those languages that can be generated by a special sub-class of CFGs called right-linear grammars.

Right-linear grammars

DEFINITION

Let $G = (V, \Sigma, R, S)$ be a CFG. We say G is a right-linear grammar if all productions in R have one of the following 2 forms:

$$A \rightarrow xB \quad \text{or} \quad A \rightarrow x$$

where $A, B \in V$ and $x \in \Sigma^*$.

(each rule contains at most **one** variable on the right of the arrow, and it must occur **after** any terminals.)

Right-linear grammars - examples

Example

CFG G_1 from last video does **not** fit that form, so it is **not** a right-linear grammar.

$$\boxed{A \rightarrow 0A1} \quad \textbf{!!}$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Right-linear grammars - examples

<u>Example</u>

The CFG $G_4 = (\{S\}, \{a,b\}, R, S)$ with R given by

$$S \rightarrow abS \mid a$$

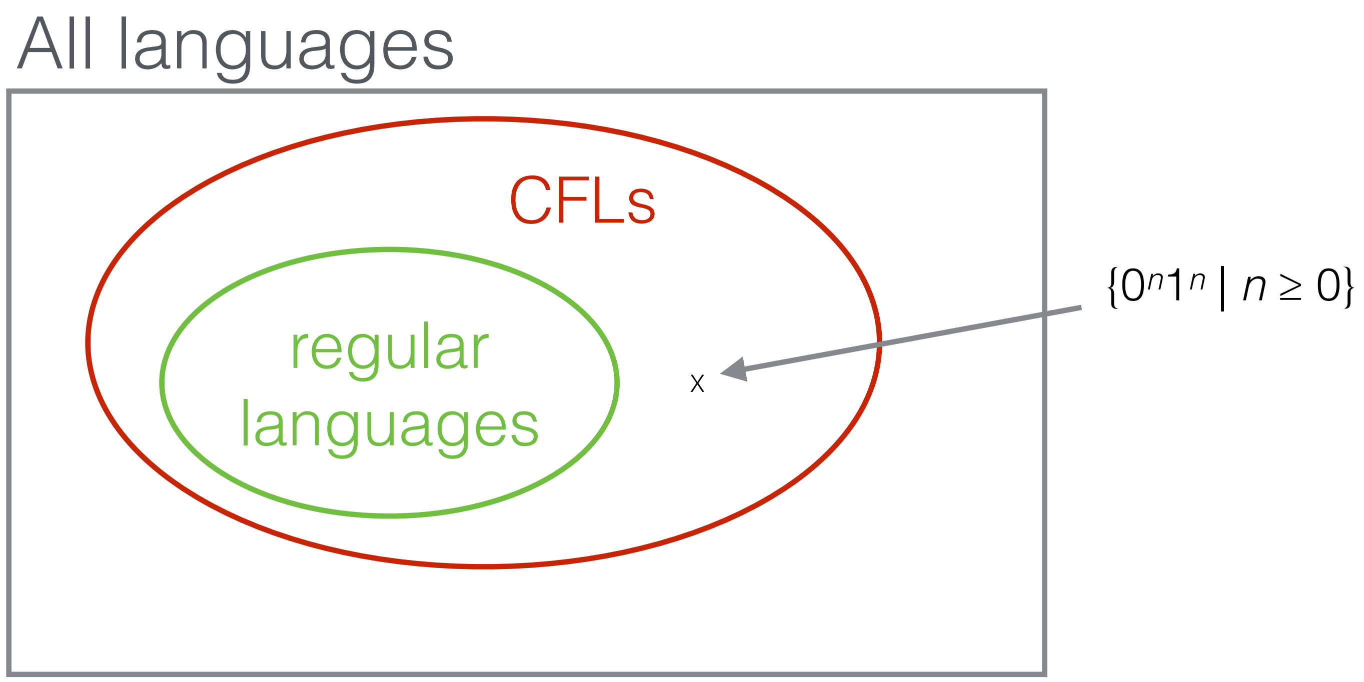
is right-linear.

<u>QUESTION</u> Which strings does G_4 generate? What is L(G_4)?

ANSWER: L(G_4) = {a, aba, ababa, abababa, etc…}, equivalently, it's the language generated by regular expression (ab)*a, or by a(ba)*.

Relation between CFLs and regular languages

The current picture:



CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

Pushdown Automata

Generators vs recognisers

- A CFG is **not** a model of computation. It is a way of *generating* strings, and the whole set of strings that a given CFG generates is the language of *G*.

- In this regard, CFGs fill the same role for CFLs as regular expressions (a*b, a(b∪c)*, etc) do for regular languages. Each regular expression provides a way to generate a set of strings.

	regular languages	**context-free languages**
Generator	regular expression	context-free grammar
Recogniser		

Generators vs recognisers

- We know regular languages have an associated model of computation, namely FAs (either DFAs or NFAs) that plays the role of recognising strings from a language.
- That is, given an FA associated to a given regular language, you feed a string to the FA and it will tell you if the string is **in** (accept) or **not in** (reject) the language.

	regular languages	**context-free languages**
Generator	regular expression	context-free grammar
Recogniser	FA	

Generators vs recognisers

- What do we have that can play the role of recogniser for CFLs?

- We know FAs won't be able to do this job, because they can only recognise regular languages, but some CFLs aren't regular (e.g. $\{0^n 1^n \mid n \geq 0\}$)

	regular languages	**context-free languages**
Generator	regular expression	context-free grammar
Recogniser	FA	**?**

Generators vs recognisers

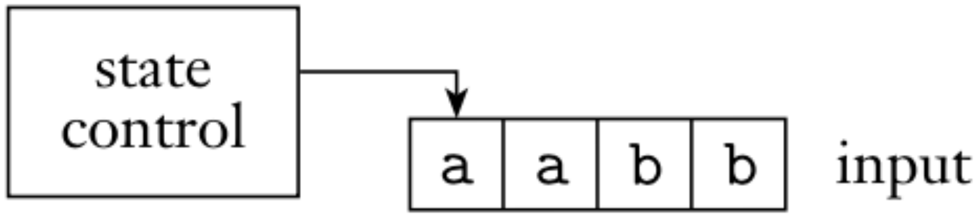
- We need a different model of computation. One that is more sophisticated than an FA.
- Such a model is the Pushdown Automaton (PDA).

	regular languages	context-free languages
Generator	regular expression	context-free grammar
Recogniser	FA	**PDA**

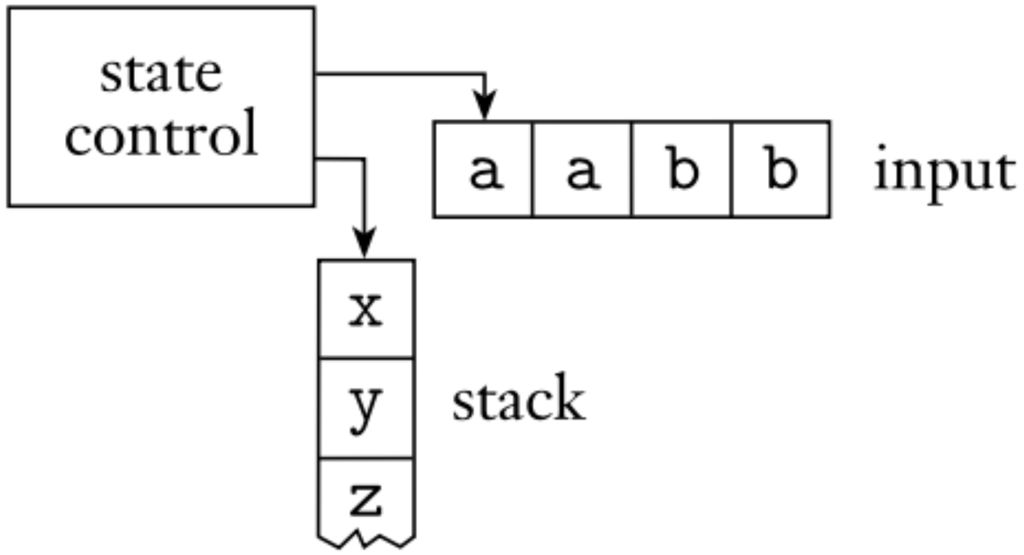
Pushdown Automata

- Pushdown automata are like NFAs, but with an additional memory component called a stack.

NFA



PDA



Pushdown Automata - The stack

- A PDA can write symbols onto the stack and read them back later.
- Writing a symbol *pushes down* all the other symbols on the stack.
- At any time, the symbol at the top of the stack can be read and removed, and then the remaining symbols move back up.
- Writing a symbol on the stack is referred to as pushing the symbol, and removing one is referred to as popping it.

Pushdown Automata - The stack

- Think of the stack as being like a spring-loaded plate dispenser in a cafeteria.
- Except the plates have symbols written on them.

A PDA that recognises $\{0^n 1^n \mid n \geq 0\}$

(informal)

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as a 1 is seen, pop a 0 off the stack for each 1 read. If reading the input is finished exactly when the stack becomes empty of 0s, **accept** the input. If the stack becomes empty while 1s remain *or* if 1s are finished while stack is non-empty *or* if 0s appear in the input following 1s, **reject** the input.

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

PDAs Recognise CFLs

PDAs:
Deterministic of Nondeterministic?

- Recall that FAs come in both a deterministic (DFAs) and a nondeterministic (NFAs) variant.

    - Deterministic - always one possible choice for what to do next at any stage during a computation.

    - Nondeterministic - sometimes several choices, leading to branching into different paths the computation can take.

PDAs: Deterministic of Nondeterministic?

- For FAs, both variants recognise the same class of languages (namely the class of regular languages), so they are equivalent in terms of expressivity.

- For PDAs too we can choose either a deterministic (DPDA) or a nondeterministic (NPDA) variant.

- **But**, unlike for FAs, these 2 are **not** equivalent - NPDAs recognise more languages than DPDAs can.

- We focus here on NPDAs, since it is they that correspond to CFLs.

PDAs - formal definition

DEFINITION

A nondeterministic pushdown automaton (or just pushdown automaton from now on) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

1. Q is a finite set called the *states.*
2. Σ is a finite set called the *input alphabet*.
3. Γ is a finite set called the *stack alphabet.*
4. δ: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$is the *transition function.*
5. q_0 is the *start state*
6. $F \subseteq Q$ is the set of *accept states* (or *final states*)

PDAs - formal definition

- As usual the heart of the definition is this bit:

  4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function.*

  (recall that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$)

- For $q \in Q$, $a \in \Sigma_\varepsilon$, $b \in \Gamma_\varepsilon$, $\delta(q,a,b)$ tells us the (set of) possible next moves the PDA makes when:
  - current state is q
  - next input symbol read is a
  - symbol read (and then popped) at top of stack is b

(*Note*: a or b could be ε, in which case PDA moves without reading input or stack.)

PDAs - formal definition

- As usual the heart of the definition is this bit:

  4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function.*

  (recall that $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$)

- The range of δ is $\mathcal{P}(Q \times \Gamma_\varepsilon)$, i.e., power set of $Q \times \Gamma_\varepsilon$, which means $\delta(q,a,b)$ returns a set

$$\{(r_1,c_1), (r_2,c_2),\ldots, (r_m,c_m)\}$$

  of possible next moves, where $(r_i,c_i) \in Q \times \Gamma_\varepsilon$, for $i = 1,\ldots,m$.

- Each (r_i,c_i) corresponds to the instruction "*change state to r_i and push c_i to the top of the stack*"

Acceptance for PDAs

A PDA $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ accepts input w if w can be written as $w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$, and there exist sequences of states $r_0,r_1,\ldots,r_m$ in Q and strings $s_0,s_1,\ldots,s_m \in \Gamma^*$ that satisfy the following:

(s_i = full stack contents at stage i)

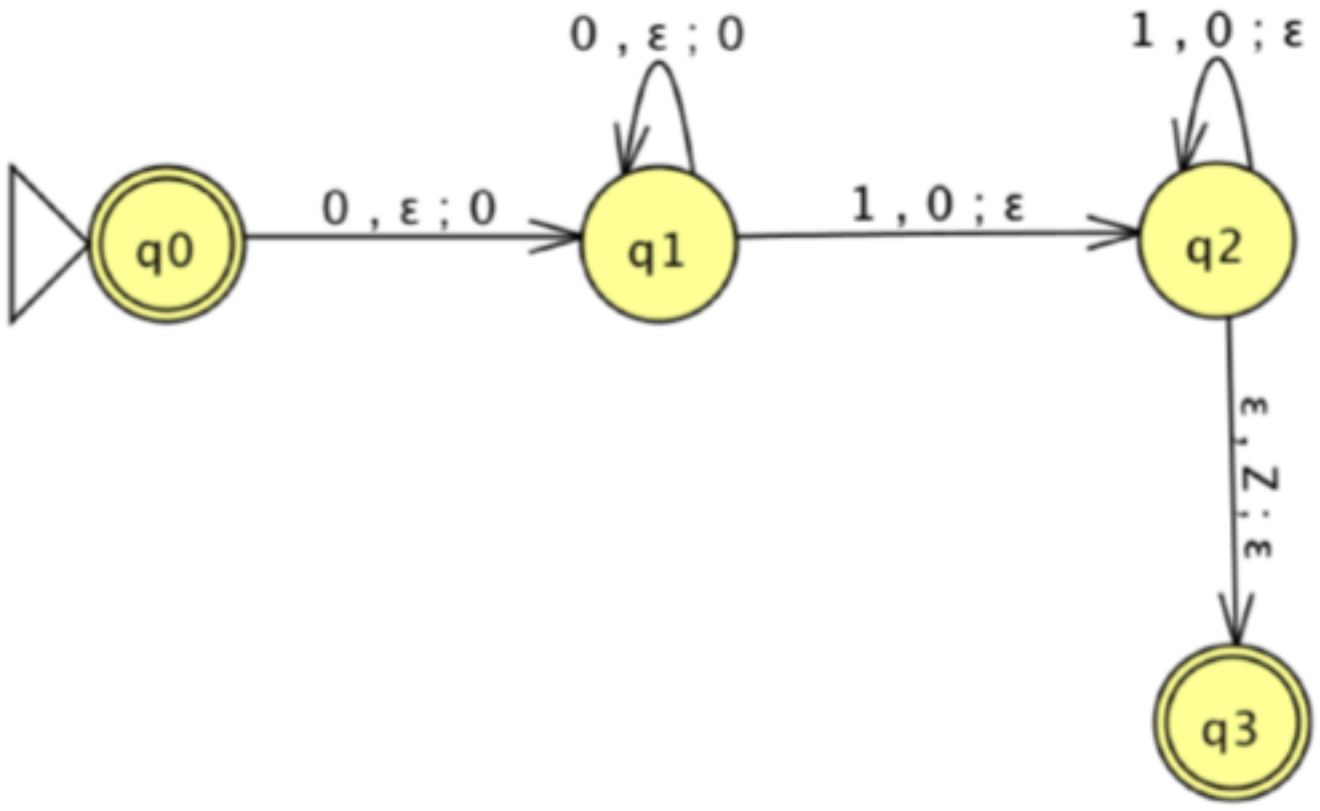
1. $r_0 = q_0$ and $s_0 = \varepsilon$. (M starts in start state and with empty stack).
2. For $i = 0,1,\ldots,m\text{-}1$, we have $(r_{i+1},b) \in \delta(r_i,w_{i+1},a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a,b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. (M moves properly according to state, stack and next input symbol.)
3. $r_m \in F$.

PDA state diagrams

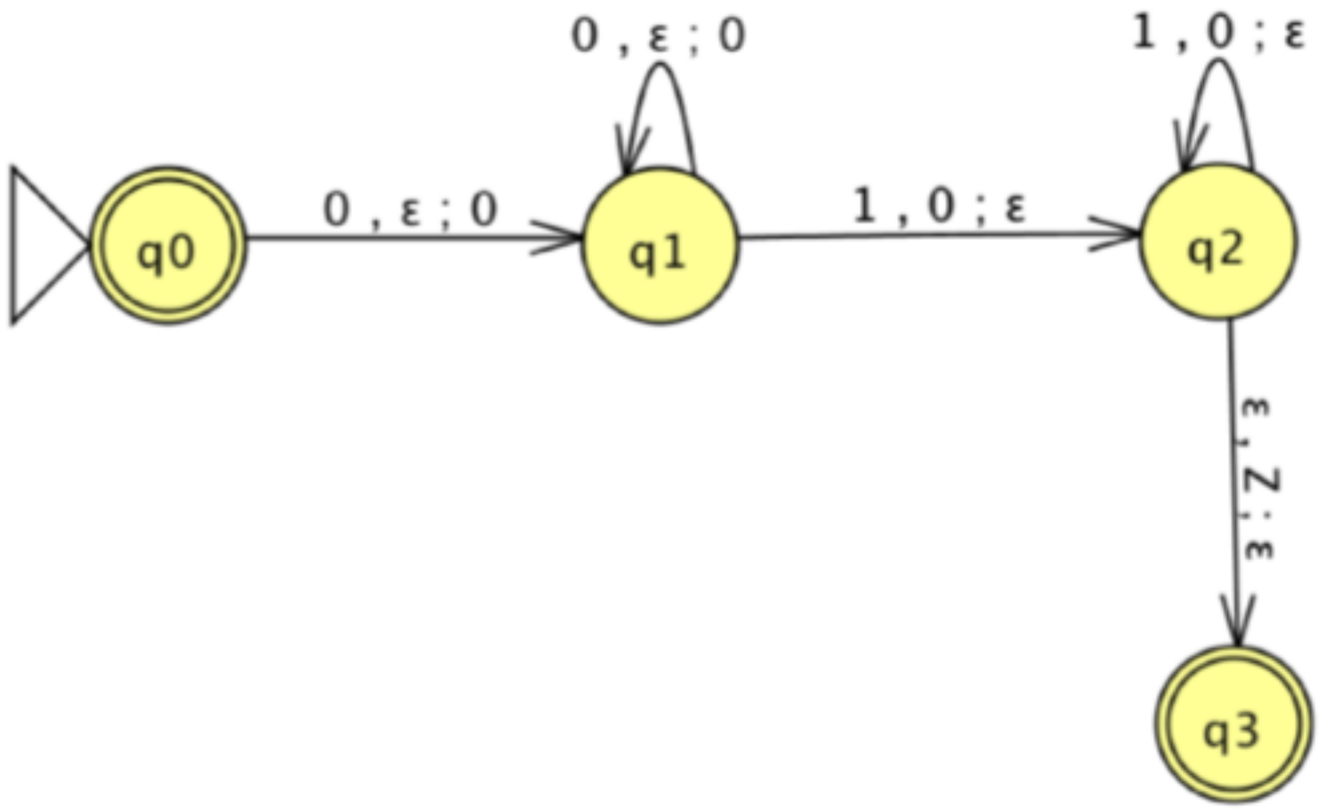
Just like FAs, PDAs can be described using their own kind of state diagram.

Example

State diagram created in JFLAP for a PDA that recognises $\{0^n1^n \mid n \geq 0\}$



PDA state diagrams in JFLAP



- Arrow from state *q* to *q'* labelled with *a,b;c* corresponds to transition $\delta(q,a,b) = (q',c)$.
- JFLAP has a special stack symbol *Z* to mark the bottom of the stack (so PDA always knows when nothing left in the stack).

PDAs recognise CFLs

For any language generated by a CFG it's possible to build a PDA that recognises it. And, the other way around, every language recognised by a PDA can be generated by some CFG.

In other words:

THEOREM
A language is context-free iff some PDA recognises it.

PROOF
See Sipser book, pages 117-124(!)