PMA1050/6853
Discrete Foundations
2004–5

Motivation

Computer systems are used

- in aircraft systems (navigation, stability, etc.)
- for air traffic control
- for the control of railway signals and junctions
- in medical systems
- in nuclear power stations

Conclusion

Software must work correctly as soon as it is installed, especially in safety–critical systems.

But in practice

Software rarely works correctly the first time it is used; reliability is only achieved after a long period of service.

How can the software development process be improved ?

- The specification should be written in a formal mathematical language.
- Quality assurance can be improved by the use of mathematical logic to verify that the system will perform as the specification requires.

1. Formal specification

- Natural language (English, French, etc.) can be ambiguous:

  A number less than two is less than one.

  Is any number a perfect square?

  but mathematics is unambiguous.

- Natural language can hide logical flaws:

  I can't solve quadratic equations when I am sober; so if I'm drunk I can.

  but mathematical logic reveals these flaws.

2. Mathematical verification

- A computer program cannot be tested exhaustively
- If we can write a formal specification in logic of what the program *should* do, we may be able to prove mathematically that our system performs to the specification.

Example : A pair of traffic lights :
The north-south and east-west lights must never be simultaneously green, and logic can be used to prove that *the specified program satisfies this and other conditions.*
But we still have to write the software in a language that will not be mathematical logic.
We then test the software against the mathematical logic.

Summary :

- Many systems are necessarily complex, and must interact with ill–understood environments, but their performance is critical for human safety.

We therefore need:

- The ability to model systems and their interactions, such that the final system operates in the required manner.
- A systematic and logical approach for enquiry that will enable general statements about systems to be made **and proved**.
- A notation that is universal, compact and easily understood for describing the behaviour of these systems.

It is these considerations, driven by the need for certainty in safety-critical systems, which make some areas of computer science very formal and abstract, much like pure mathematics; sometimes even more so.
This course

- introduces the notation and ideas of **Set Theory**,
- studies **Mathematical Proof**,

- introduces **Propositional Logic** — the most basic type of formal reasoning, and
- introduces the theory of **Formal Languages and Abstract Automata**.

Course Contents

- Motivation
- Set Theory
- Functions
- Proof
- Propositional Logic
- Finite State Machines
- Languages and Grammars

Set Theory

Set theory provides an infrastructure that allows both description and reasoning about systems and their behaviour.
It is

- simple
- concise
- unambiguous
- accurate

Why do we need to know about sets ?

- A computer program may be thought of as a function that maps one set of data (the input) to another (the output).
- The use, understanding and construction of formal specifications requires the use of sets.
- The concepts that underlie relational databases involve set theory.
- Set theory is closely connected with propositional logic and Boolean algebra.

It is often necessary to write programs involving data other than numbers or geometric figures, and to prove statements about these entities.

- When considering the safety of a set of traffic lights, it is necessary to prove statements about the colours red, amber and green.
- A word in a word processor is represented by a string of characters. The manipulation of words therefore reduces to the manipulation of strings of characters.

Sets, and strings, are data types of many programming languages.

Definitions :

1. A *set* is a well-defined collection of objects.
2. The objects of the set are called its *elements*.

Notation for the specification of sets :

1. Specify a set by its members

   $$A = \{a, 2, red\}.$$

2. Specify a set by the properties that define its elements (comprehension)

   $$B = \{x : x \text{ is an integer, } x > 0\}.$$

   The colon (sometimes | is used) is read as 'such that'.

When defining sets by listing their elements:

1. order is irrelevant. Thus

   $$\{a, 2, red\} \quad \text{and} \quad \{2, a, red\}$$

   are the same set.

2. there is no concept of multiple occurrence of elements in a set. Thus

   $$\{2, a, red, 2, 2, a, 2\} \quad \text{and} \quad \{a, 2, red\}$$

   are the same set.

We write $a \in A$ to denote that a is an element of the set A.
The only meaningful question we can ask is about set membership.

Some standard sets

Some commonly used sets are

1. $B = \{true, false\}$ is the set of *Boolean values*.

2. $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ is the set of *natural numbers.*
3. $\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \dots\}$ is the set of *integers.*
4. $\mathbb{R}$ is the set of all real numbers.

The sets $\mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$ are infinite.

Sets specified by comprehension.

1.

$$\begin{aligned} \{x : x \text{ is an integer greater than } 2\} &= \{x \in \mathbb{Z} : x > 2\} \\ &= \{3, 4, 5, \dots\}. \end{aligned}$$

2.

$$\{x \in \mathbb{Z} : x^2 \le 10\} = \{-3, -2, -1, 0, 1, 2, 3\}.$$

3.

$$\{x \in \mathbb{R} : x^2 \le 0\} = \{0\}.$$

Subsets

If every element of set A is also an element of another set B, then A is a *subset* of B, $A \subseteq B$. If $A \subseteq B$ and $A \ne B$, we say A is a *proper subset* of B, $A \subset B$, (c.f. $x < y$ and $x \le y$ for numbers x, y).
Example If

$$A = \{a, 2, \mathit{red}\} \text{ and } B = \{a, 2, \mathit{red}, \mathit{green}\}$$

then $A \subseteq B$. However, $A \ne B$ because $\mathit{green} \in B$ but $\mathit{green} \notin A$, so we can also make the stronger statement $A \subset B$.

Example If

$$A = \{w, x, y, z\} \text{ and } B = \{y, w, z, x\}$$

then $A \subseteq B$ and $B \subseteq A$ because A and B have exactly the same elements.

Equality of sets

Generally, if

$$A \subseteq B \text{ and } B \subseteq A, \text{ then } A = B.$$

This follows because every element of A is an element of B,
and every element of B is an element of A.

If at least one element of A does not belong to B, then A is not contained in B, $A \not\subseteq B$.
Example If

$$A = \{1, 3, 4, 5, 8, 9\}, \qquad B = \{1, 2, 3, 5, 7\}, \qquad C = \{1, 5\}$$

then $C \subset A$, $C \subset B$ and $B \not\subseteq A$. □

Example Let A be the set of all people who live in a region and have a telephone, and let B be the set of all these people who are listed in the telephone directory. Then

1. If nobody is ex–directory, then $A = B$
2. If there is at least one person who is ex–directory, then $B \subset A$

In either case, we can say $B \subseteq A$.

The Universal Set

The members of all sets under investigation usually belong to a larger set called the *universal* or *fixed* set, denoted by U.

- The set U in plane geometry consists of all points in the plane
- The set U in studies of human population consists of all the people in the world
- The set U is $\mathbb{N}$ when counting in discrete domains
- The set U is $\mathbb{R}$ when counting and measuring in continuous domains
- The set U is *Alphabet* when dealing with sets of characters

The Empty Set

For a given set U and property P, there may not be any elements of U that satisfy property P.

$$S = \{x : x \text{ is a positive integer, } x^2 = 3\}$$

has no elements.
The set with no elements is called the *empty* or *null* set and is denoted by $\emptyset$.
Properties :

- The empty set is unique, because a set is defined by saying what its elements are.
- The empty set $\emptyset$ is a subset (not, generally, an element) of every set A: i.e. $\emptyset \subseteq A$.

Examples

1. $\emptyset = \{x : x \neq x\}$
2. $\emptyset = \{n : n \in \mathbb{N} \text{ and } n > n\}$
3. $\emptyset =$ The set of rivers where water flows upstream
4. $\emptyset =$ The set of African countries that lie north of the Mediterranean
5. $\emptyset =$ The set of European countries where rice is the staple diet

The Power Set

Given a set A, we may be interested in some of its subsets, and thus we need to consider a *set of sets*, (you might say *class of sets* or a *collection of sets.*)

Example Let $A = \{1, 2, 3, 4\}$ and let B be the set of subsets of A that contain exactly three elements of A.

$$B = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}\}$$

Let C be the set of subsets of A that contain 2 and two other elements of A

$$C = \{\{1, 2, 3\}, \{1, 2, 4\}, \{2, 3, 4\}\}$$

Thus C is a subset of B, since every element of C is also an element of B.

For every set A, we maybe interested in the set of all subsets of A. This is called the *power set of A* and denoted by $P(A)$.

Example Let $A = \{a, b, c\}$. Then
$P(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{c, a\}, \{a, b, c\}\}$
Notes :

- The empty set $\emptyset$ is a member of $P(A)$.
- The set A is a member of $P(A)$.
- The number of elements of $P(A)$ is $2^3 = 8$.

If the set A has n elements, (we say A has *cardinality n*), then the number of elements in $P(A)$ is 2^n.

Set operations

1. **Union** $A \cup B$ is the set of all elements that belong to A or B :
$$A \cup B = \{x : x \in A \text{ or } x \in B\}.$$
'or' is used in the sense of and/or ('vel' in Latin).
2. **Intersection** $A \cap B$ is the set of all elements that belong to both A and B :
$$A \cap B = \{x : x \in A \text{ and } x \in B\}.$$
3. **Set difference** $A \setminus B$ is the set of all elements that are in A but are not in B :
$$A \setminus B = \{x : x \in A \text{ and } x \notin B\}.$$
4. **Complement** A^c is the set of all elements that are in U but are not in A :
$$A^c = \{x : x \in U \text{ and } x \notin A\}.$$

Example
Define the sets A, B and U by

$$A = \{1, 2, 3, 4\},\ B = \{3, 4, 5, 6\},\ U = \{1, 2, 3, \dots\}.$$

Then

$$A \cup B = \{1, 2, 3, 4, 5, 6\}, \qquad A \cap B = \{3, 4\},$$

$$A \setminus B = \{1, 2\}, \qquad A^c = \{5, 6, 7, 8, \dots\} \qquad \square.$$

Note the distinction :

- $\in$ denotes membership
- $\subseteq$ denotes inclusion

Example
The following statements are true :
$\mathbb{N} \subseteq \mathbb{Z},\ \mathbb{N} \subset \mathbb{Z},\ \{1, 3, 7\} \subseteq \mathbb{N},\ \{1\} \subseteq \mathbb{R},\ 1 \in \mathbb{R},$
$\emptyset \subset \mathbb{N}, \quad \emptyset \subset \{0\}, \quad -5 \in \mathbb{Z}, \quad \emptyset \subseteq \emptyset$
The following statements are false :
$1 \subset \mathbb{R}, \qquad \{0\} \subset \emptyset, \qquad \{3, 4\} \in \mathbb{N}$

Venn diagrams

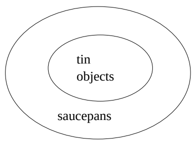
- A Venn diagram is a pictorial representation of sets by subsets of points in the plane.
- The universal set U is represented by the interior of a rectangle, if necessary, and the other sets are represented by disks lying within the rectangle.
- Many verbal statements can be translated into equivalent statements about sets which can be described by diagrams.

- Thus, in very simple cases only, Venn diagrams may be used to determine the validity of an argument.

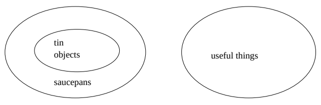
Example Consider the statements S_1, S_2 and S_3 and the conclusion C:

1. S_1 : My saucepans are the only things that I have that are made of tin.
2. S_2 : All your presents are useful.
3. S_3 : None of my saucepans is of the slightest use.
4. C : The presents that you give me are not made of tin.

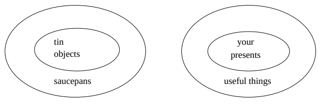
The Venn diagram for S_1 is



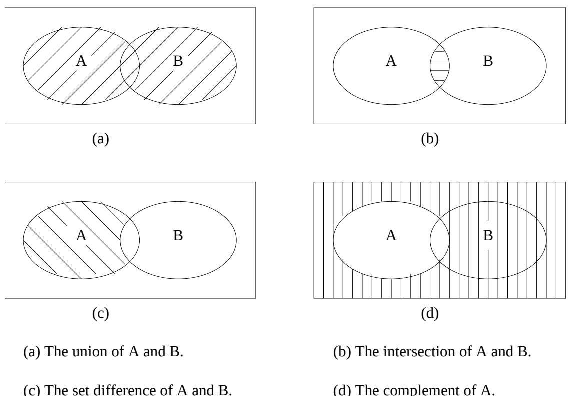
The Venn diagram for S_3 is



The Venn diagram for S_2 is



Thus the conclusion is true.



(a) The union of A and B. (b) The intersection of A and B.

(c) The set difference of A and B. (d) The complement of A.

The laws of set algebra

1. Idempotent Laws :

   $A \cup A = A \qquad A \cap A = A$

2. Associative Laws :

   $(A \cup B) \cup C = A \cup (B \cup C)$
   $(A \cap B) \cap C = A \cap (B \cap C)$

3. Commutative Laws :

   $A \cup B = B \cup A \qquad A \cap B = B \cap A$

4. Distributive Laws :

   $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

   $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

5. Identity Laws :

   $A \cup \emptyset = A \qquad A \cap U = A$
   $A \cup U = U \qquad A \cap \emptyset = \emptyset$

6. Complementary Laws :

   $(A^c)^c = A \qquad A \cup A^c = U$

   $A \cap A^c = \emptyset \qquad U^c = \emptyset \qquad \emptyset^c = U$

7. De Morgan's Laws :

   $(A \cup B)^c = A^c \cap B^c \qquad (A \cap B)^c = A^c \cup B^c$

Example Prove

$$(A \cup B)^c = A^c \cap B^c$$

Proof : Two stages

1. $(A \cup B)^c \subseteq A^c \cap B^c$
2. $A^c \cap B^c \subseteq (A \cup B)^c$

1. If $x \in (A \cup B)^c$ then $x \notin (A \cup B)$. Thus $x \notin A$ and $x \notin B$; i.e. $x \in A^c$ and $x \in B^c$. Hence $x \in A^c \cap B^c$.
2. Try proving this yourselves.

Example Prove

$$(A \cup B)^c = A^c \cap B^c$$

Proof : Two stages

1. $(A \cup B)^c \subseteq A^c \cap B^c$
2. $A^c \cap B^c \subseteq (A \cup B)^c$

1. If $x \in (A \cup B)^c$ then $x \notin (A \cup B)$. Thus $x \notin A$ and $x \notin B$; i.e. $x \in A^c$ and $x \in B^c$. Hence $x \in A^c \cap B^c$.
2. If $x \in A^c \cap B^c$, then $x \in A^c$ and $x \in B^c$. Thus $x \notin A$ and $x \notin B$; i.e. $x \notin A \cup B$. Thus $x \in (A \cup B)^c$.
The result follows from the combination of the two stages. $\square$

Use the laws of set algebra to simplify set expressions.

Example

$$\begin{aligned}
A \setminus (A \setminus B) &= A \cap (A \cap B^c)^c \\
&\qquad (A \setminus B = A \cap B^c \text{ twice}) \\
&= A \cap (A^c \cup (B^c)^c) \qquad \text{(de Morgan)} \\
&= A \cap (A^c \cup B) \qquad \text{(complement)} \\
&= (A \cap A^c) \cup (A \cap B) \qquad \text{(distribution)} \\
&= \emptyset \cup (A \cap B) \qquad \text{(complement)} \\
&= A \cap B \qquad (U \text{ and } \emptyset) \qquad \square
\end{aligned}$$

Example Prove that $(A \cap B) \subseteq A \subseteq (A \cup B)$.

1. Since every element in $A \cap B$ is in A and B, it follows that if $x \in (A \cap B)$, then $x \in A$.
2. Hence $(A \cap B) \subseteq A$.
3. Furthermore, if $x \in A$, then $x \in (A \cup B)$, and so $A \subseteq (A \cup B)$.
4. The combination of these results yields $(A \cap B) \subseteq A \subseteq (A \cup B)$.

Functions

A *function* (or *mapping*) f is, loosely speaking, a 'rule' that assigns to each element x in a set A one element y of another set B.
The set A is called the domain of the function f.
The set B is called the co–domain of the function f.

Example
The rule which assigns to each real number x the real number $\sin x$ is a function with domain $\mathbb{R}$ and co-domain $\mathbb{R}$. $\square$

Example
With a little more thought, we can say that sin is a function with domain $\mathbb{R}$ and co-domain the set of all real numbers between -1 and +1 (inclusive). $\square$

Example
A more complicated rule is to map a real number x to $+x$ if $x \geq 0$ and $-x$ if $x < 0$. The 'rule' does not have to be a single formula. This is the definition of the *modulus* function, with domain $\mathbb{R}$ and co-domain $\mathbb{R}$ (or co-domain the non-negative real numbers). $\square$

Example
A function does not have to be given by a simple rule. The Sheffield telephone directory is a rule that ascribes to each person exactly one telephone number. It defines a function whose domain is the set of people listed therein, with co-domain the set of all seven-digit sequences. $\square$

Example
Since nobody can have more than one birthday, the mapping 'birthday' is a function with domain, say, this lecture class and co-domain the days of the year. $\square$

Notation :

1. $f : A \to B$

   denotes that the function f maps the set A into the set B.

2. $f : x \mapsto y$

   denotes that the function f maps the element x of A to the element y of B. We can also express this by writing

$$y = f(x).$$

3. We say $f(x)$ is the *image* of x under the function f or the *value* of f at x.

Example
Let f be the function from $\mathbb{R}$ to $\mathbb{R}$ (i.e. $f : \mathbb{R} \to \mathbb{R}$) that maps every real number to its square. This function may be written as

$$f(x) = x^2 \qquad \text{or} \qquad f : x \mapsto x^2$$

Example
Let $A = \{1, 2, 3, 4\}$ and $B = \{a, b, c, d, e\}$. We may define a function f with domain A and co-domain B by

$$f(1) = a, \qquad f(2) = c, \qquad f(3) = e, \qquad f(4) = a.$$

Note :

1. Every element of the domain must be mapped to an element of the co-domain.

2. Some elements of the co-domain may not be the images of any elements of the domain.

3. No element of the domain is mapped to more than one element of the co-domain.

4. An element of the co-domain may be the image of more than one element of the domain.

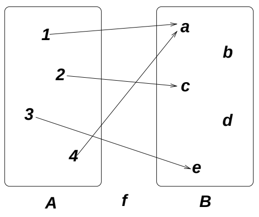
The subset of the co–domain that consists of all those elements that are the mapping of elements in the domain is called the *range* of f :

$$range(f) = \{f(a) : a \in A\}$$

The range is sometimes written $f(A)$; generally, for $D \subseteq A$ we write

$$f(D) = \{f(x) : x \in D\}.$$

Example
For the sets in the previous example



$domain(f) = \{1, 2, 3, 4\}$
$codomain(f) = \{a, b, c, d, e\}$
$range(f) = \{a, c, e\}$ □

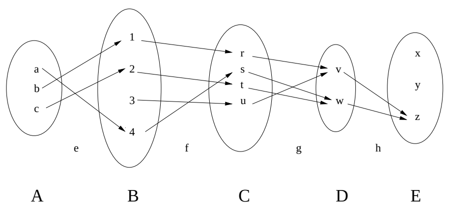
One–to–one, onto and invertible functions

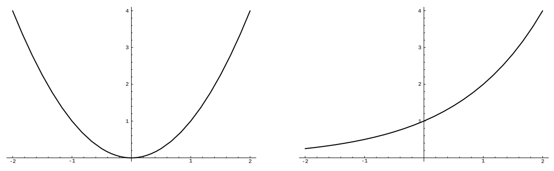
Definitions :

1. A function $f : A \to B$ is one–to–one or *injective*, written as 1–1, if distinct elements in the domain A have distinct images. This is equivalent to the requirement that f is one–to–one if $f(a) = f(a')$ implies that $a = a'$.

2. A function $f : A \to B$ is onto or *surjective* if each element of B is the image of an element in A. Thus $f : A \to B$ is onto if the image of f is the entire co–domain, that is, $f(A) = B$.

3. A function $f : A \to B$ is *invertible* or *bijective* if the inverse relation is a function from B to A.





Example
Consider the functions $e : A \to B$, $f : B \to C$, $g : C \to D$ and $h : D \to E$.
e and f are both 1–1, but g and h are not 1–1.
f and g are both onto, but e and h are not onto.
h is neither 1–1 nor onto.

A function f is *invertible* if and only if it is both 1–1 and onto.

1. If $f : A \to B$ is both 1–1 and onto, then f is a *one–to–one correspondence* between A and B.

2. In this case, each element of A maps to a distinct element of B, and vice–versa.

From the previous example, only f is invertible.

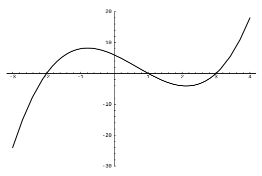
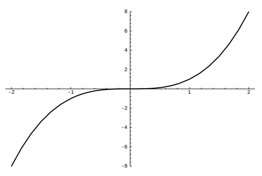
Example
Consider a telephone directory.

- Let A be the set of residents who are listed in the local telephone directory and let B be the set of all the telephone numbers in the area that is covered. If some residents are ex–directory, then the mapping from A to B is 1–1 but not onto.

- Let A be set of employees of a company and let B be the set of their telephone extensions. Assuming that all the employees are listed and that every person has his/her own extension, then the mapping from A to B is invertible.□

$e : \mathbb{R} \to \mathbb{R}; e(x) = x^2$ is neither 1–1 nor onto.
$f : \mathbb{R} \to \mathbb{R}; f(x) = 2^x$ is 1–1 and not onto.

$g(x) = x^3 - 2x^2 - 5x + 6$ is onto but not 1–1.
$h(x) = x^3$ is 1–1 and onto.
Note: *increasing* functions are 1–1

Example

Show that the function $f : \mathbb{R}\setminus\{0\} \to \mathbb{R}\setminus\{1\}$, given by $f(x) = (x+1)/x$, is 1–1 and onto.

A function is 1–1 if distinct elements in the domain have distinct images. Thus if it is assumed that the images of two elements x_1 and x_2 in the domain of f are equal,

$$f(x_1) = f(x_2),$$

implies that $x_1 = x_2$, then the function f is 1–1. But

$$\frac{x_1+1}{x_1} = \frac{x_2+1}{x_2}$$

implies (on multiplying through by x_1x_2 that

$$x_2(x_1+1) = x_1(x_2+1),$$

whence $x_1 = x_2$.

To check that f is onto, it is necessary to show that the equation

$$y = \frac{x+1}{x}$$

has a solution $x \in \mathbb{R}$ for every $y \in \mathbb{R}\setminus\{1\}$. The equation is equivalent to, successively,

$$y = 1 + \frac{1}{x}, \qquad y - 1 = \frac{1}{x}, \qquad \frac{1}{y-1} = x.$$

It follows that f is onto: for every value of y in $\mathbb{R}\setminus\{1\}$, there is a unique solution x.

Since f is 1–1 and onto, it is also invertible. In fact, the inverse function is

$$y \mapsto \frac{1}{y-1}.$$

Products of Sets

Given two sets A and B, their **Cartesian product** $A \times B$ is the set

$$A \times B = \{(a,b) : a \in A,\ b \in B\},$$

the set of all **ordered pairs** of elements: the first from A, the second from B.

2 **Examples**

- $\mathbb{R}\times\mathbb{R}$, usually written $\mathbb{R}^2$, is the set of all ordered pairs (x,y) of real numbers; it can be thought of as the set of all points in the plane.
- If $A = \{1,2\}$ and $B = \{a,b,c\}$, then
$$A \times B = \{(1,a),(1,b),(1,c),(2,a),(2,b),(2,c)\}.$$
- $A \times \emptyset = \emptyset,\ \ \emptyset \times A = \emptyset.$

Note that we are talking about **ordered** pairs, like pairs of 'coordinates' of points in the plane, so generally $A \times B \neq B \times A$. In the second example above,

$$B \times A = \{(a,1),(b,1),(c,1),(a,2),(b,2),(c,2)\}.$$

Products of more than two sets are handled in a similar way:

$$A \times B \times C = \{(a,b,c) : a \in A,\ b \in B,\ c \in C\}.$$

Examples

- $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$, usually written $\mathbb{R}^3$, is the set of all ordered triples (x,y,z) of real numbers; it can be thought of as the set of all points 3-dimensional space.
- If $A = \{1,2\}$ and $B = \{a,b,c\}$, then
$$\begin{aligned} A \times B \times A &= \{(1,a,1),(1,b,1),(1,c,1),(2,a,1),\\ &\quad (2,b,1),(2,c,1),(1,a,2),(1,b,2),\\ &\quad (1,c,2),(2,a,2),(2,b,2),(2,c,2)\}. \end{aligned}$$

Relations

A **relation** on a set is something which may or may not be true of an ordered pair of elements of the set.

Examples

- the relation $\leq$ on the set $\mathbb{R}$ of all real numbers: for example $1 \leq 2$ is true; $2 \leq 1$ is not.
- the relation of equality on the set of characters $A = \{x,y,z\}$: $x = x$, $y = y$ and $z = z$ are true; $x = y$ is not.
- the relation $y = x^2$ on the set $\mathbb{R}$; let us call this relation R, so that we write xRy to mean $y = x^2$; then $1R1$, $2R4$ and $(-3)R9$ are true, but $2R2$ and $6R3$ are not.

A relation R on a set A defines a subset of $A \times A$, namely

$$\{(x, y) \in A \times A : xRy\}.$$

In a formal development of set theory, relations are defined to be subsets of $A \times A$.

Example

If R is a relation on $\mathbb{R}$ defined by a function $f : \mathbb{R} \to \mathbb{R}$,

$$xRy \text{ if and only if } y = f(x),$$

then the associated subset of $\mathbb{R} \times \mathbb{R}$ is

$$\{(x, y) : y = f(x)\}.$$

This set is called the **graph** of f

When is a relation R on a set A defined by a function $f : A \to A$ using

$$xRy \text{ if and only if } y = f(x)?$$

A necessary and sufficient condition is:

for every $x \in A$ there is **one and only one** $y \in A$ with xRy.

Properties of relations

A relation R on a set A is said to be:

- **reflexive** if xRx for every $x \in A$;
- **symmetric** if xRy implies yRx for every $x, y \in A$;
- **transitive** if xRy and yRz imply xRz for every $x, y, z \in A$.

Examples

- Let A be the plane and say pRq if the distance between the points p and q is less than 1. Then R is reflexive and symmetric, but not transitive.
- Let $A = \mathbb{R}$ and say xRy if and only if $x - y$ is an integer. Then R is reflexive, symmetric and transitive.
- Let $A = \mathbb{R}$ and R be $\leq$. Then R is reflexive and transitive, but not symmetric.

Equivalence Relations

A relation R on a set A is an **equivalence relation** if it is reflexive, symmetric and transitive.

Example

- Let $A = \mathbb{R}$ and say xRy if and only if $x - y$ is an integer.
- Let $A = \mathbb{Z}$ and say xRy if $x - y$ is even.

Theorem (The only theorem about equivalence relations!)

If R is an equivalence relation on a set A, then A splits as a union of disjoint subsets (**equivalence classes**) so that for $x, y \in A$, we have xRy if and only if x and y belong to the same equivalence class.

An example will make this clear.

Example

Let $A = \mathbb{Z}$ and say xRy if $x - y$ is even.

Then

$$A = E \cup O$$

where E denotes the set of all even integers and O denotes the set of all odd integers. These sets are disjoint ($E \cap O = \emptyset$), and xRy if and only if x and y are either both even or both odd.

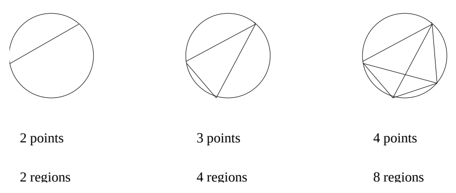
Application

We shall be using the language of equivalence relations to describe a situation when a machine can be in different but indistinguishable states. The relation of indistinguishability is an equivalence relation. Lumping indistinguishable states together into equivalence classes will show us how to construct a simpler machine, with one state corresponding to each equivalence class in the original, which does the same job. (Don't worry if this paragraph doesn't make sense yet — it will later.)

Mathematical enquiry by guesswork

Example

Consider a circle with n points in general position on its circumference, joined by straight lines:



number of points	number of regions
2	2
3	4
4	8
5	16

We can guess that the next term will be 31, but this is just a guess, not a proof. Others might make other guesses.

Proving the first few cases of a theorem, even the first million cases, does not prove the theorem.

Proof

Types of Proof :

- Direct proof
- Proof by exhaustion
- Proof by contradiction
- Proof by induction

1. Direct proof

This method of proof proceeds by straightforward manipulation of equations and formulae to arrive at the desired result.

Example
Theorem : Let a, b, c be real numbers. If $a.c = b.c$, prove that $a = b$.
Proof : Let $a.c = b.c$.
Multiply both sides of the equation by $\frac{1}{c}$:

$$a.c.\frac{1}{c} = b.c.\frac{1}{c}$$

and thus $a.1 = b.1$, or $a = b$.

What happens if $c = 0$?

The proof fails because $\frac{1}{0}$ is not a finite number.

So we rephrase the statement as

Let a, b, c be real numbers. If $a.c = b.c$ and $c \neq 0$, then $a = b$. □

2. Proof by exhaustion

This method of proof examines every possibility and establishes the truth of the statement in each case.
It can only be used when there is a finite number of situations to consider.

Example
Theorem : There do not exist integers a and b that satisfy $a^2 = 5b + 2$.
Proof : It must be shown that $a^2 - 2$ is not divisible by 5.
Every integer a can be written in the form

$$a = 5c + r, \qquad 0 \leq r < 5.$$

Then

$$a^2 - 2 = 25c^2 + 10cr + r^2 - 2 = 5\left(5c^2 + 2cr\right) + r^2 - 2.$$

The first term on the right hand side is divisible by 5, so it suffices to check that the second term, $r^2 - 2$, is not divisible by 5, and we only need to do this for $r = 0, 1, 2, 3, 4$:
$0^2 - 2 = -2$, $1^2 - 2 = -1$, $2^2 - 2 = 2$, $3^2 - 2 = 7$, $4^2 - 2 = 14$.

3. Proof by contradiction

This method of proof assumes that the converse of the desired result is true, and proceeds by showing that a logical inconsistency necessarily follows.

Example
Theorem : If a is a positive integer and a^2 is even, then a is even.
Proof : Assume the contrary, that is, a is odd. Thus a can be written in the form $a = 2r + 1$ where r is any integer. Hence

$$a^2 = 4r^2 + 4r + 1 = 4\left(r^2 + r\right) + 1$$

We are told that a^2 is even but this equation shows that this cannot be the case. It follows that the assumption that a is odd is incorrect, and a must therefore be even.

Example
Theorem : The equation $ax + b = c, a \neq 0$, has exactly one solution.
Proof : Clearly the equation has one solution. Assume that it has another solution, and let the two solutions be x_1 and x_2. Thus

$$\begin{aligned} ax_1 + b &= c \\ ax_2 + b &= c \end{aligned}$$

Subtract

$$a\left(x_1 - x_2\right) = 0$$

and since $a \neq 0$, it follows that $x_1 = x_2$.
Thus the assumption that the equation has two distinct solutions leads to the conclusion that the two solutions are the same, that is, the equation has exactly one solution. □

4. Proof by induction

This method of proof is useful for the establishment of the truth of propositions that involve an infinite list of cases, with an easy connection between each case and either the preceding one, or at least some earlier cases.

Example

1. The sum of the first n positive integers is $\frac{n(n+1)}{2}$.
2. If $n \geq 14$, then n can be written in the form

   $$n = 3a + 8b$$

   for some non-negative integers a and b.
3. Any string of n characters can be decomposed into two strictly smaller strings of characters, preserving the order of the characters, in $n-1$ ways. $\square$

The principle of mathematical induction.

Let $P(n)$ be a **proposition** involving the positive integer n.
If we know

1. $P(1)$ is true;
2. for every $n \geq 1$, $P(n)$ implies $P(n+1)$;

then $P(n)$ is true for all $n \geq 1$.

The number 1 here can be replaced throughout by 0, or by any integer.

The steps involved in induction are :

1. The **induction base**: prove the statement for $n = 1$ (or whatever your starting value is)
2. The **induction step**: assuming the result for some arbitrary $n \geq 1$, (the **induction hypothesis**) deduce the same result but with $n+1$ in place of n.
3. **Conclusion** 'By mathematical induction, the result holds for all $n \geq 1$'.

Example
Theorem : The sum of the first n positive integers is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
Proof : Note the shorthand notation. We write

$$1 + 2 + 3 + \cdots + n \text{ as } \textstyle\sum_{i=1}^{n} i.$$

Induction base. Clearly, the sum of the first positive integer is 1, and

$$\textstyle\sum_{i=1}^{1} i = \frac{1(1+1)}{2} = 1$$

and thus the theorem is true for $n = 1$.

Induction step. We now assume that the theorem is true for a given arbitrary value of n, and show that it is true for $n+1$. Thus we assume:
Induction hypothesis: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
Then

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= \sum_{i=1}^{n} i + (n+1) \\
&= \frac{n(n+1)}{2} + (n+1) \quad \text{(by Induction Hypothesis)} \\
&= \frac{n(n+1) + 2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2} \\
&= \frac{(n+1)((n+1)+1)}{2}.
\end{aligned}$$

We have

$$\sum_{i=1}^{n+1} i = \frac{(n+1)((n+1)+1)}{2},$$

which is the original formula with $(n+1)$ in place of n.
This completes the induction step.
Conclusion: By mathematical induction, the result holds for all $n \geq 1$.

Example
Theorem : $5^n - 4n - 1$ is divisible by 16 for all integers $n \geq 1$.
Proof : Define

$$S(n) = 5^n - 4n - 1$$

Induction base. For $n = 1$ we have $S(1) = 0$ which is trivially divisible by 16.
Induction step. We now assume that the theorem is true for a given arbitrary value of n, and show that it is true for $n+1$. Thus we assume:
Induction hypothesis: $S(n)$ is divisible by 16, and we try to show that $S(n+1)$ is divisible by 16.

Now

$$\begin{aligned}
S(n+1) &= 5^{n+1} - 4(n+1) - 1 \\
&= 5 \times 5^n - 4n - 5 \\
&= 5 \times 5^n - 5 \times 4n + 4 \times 4n - 5 \\
&= 5(5^n - 4n - 1) + 16n \\
&= 5S(n) + 16n.
\end{aligned}$$

By the Induction Hypothesis, $S(n)$ is divisible by 16; therefore $S(n+1)$ is divisible by 16. The induction step is proved.

Conclusion: By mathematical induction, the result holds for all $n \geq 1$.

Example

Theorem : If $n \geq 14$, then n can be written in the form

$$n = 3a + 8b,$$

where $a, b \geq 0$, and n, a, b are all integers.

Proof : Notice that our starting point here is $n = 14$

Induction base. For $n = 14$ we have $14 = 6+8$ so n can be written in the form $n = 3a + 8b$ by making $a = 2$ and $b = 1$.

Induction step. We now assume that the theorem is true for a given arbitrary value of $n \geq 14$, and show that it is true for $n + 1$. Thus we assume:

Induction hypothesis: $n \geq 14$ and n can be written in the form $n = 3a + 8b$, where $a, b \geq 0$.

We must show that there are integers $a', b' \geq 0$ such that $n + 1 = 3a' + 8b'$. By the induction hypothesis,

$$n + 1 = 3a + 8b + 1 = 3(a + 3) + 8(b - 1),$$

so the desired result for $n+1$ follows, with $a' = a+3$ and $b' = b - 1$, provided that $b \geq 1$ (so that $b' \geq 0$).

We need to consider separately the case $b = 0$. In this case, $n \geq 15$, since we know that for $n = 14$ we have $b = 1$. Then $3a = n \geq 15$, so $a = 5 + c$ for some integer $c \geq 0$. Hence

$$n + 1 = 3a + 1 = 3(5 + c) + 1 = 3c + 8.2,$$

and the desired result for $n+1$ holds, with $a' = c$ and $b' = 2$. This completes the induction step.

Conclusion: By mathematical induction, the result holds for all $n \geq 14$.

It is interesting to note that the only numbers less than 14 that can be written in the form $3a + 8b$ are $3, 6, 8, 9, 11$ and 12.

Recursive definitions and procedures

Recursion is a powerful method for the solution of many problems. It breaks down a large problem into several smaller *identical* problems.

Example

1. Consider a problem P_1, whose solution could be calculated if the answer to a problem P_2 is known, where P_2 is a smaller instance of P_1.
2. The solution to P_2 requires the answer to a problem P_3, which is a smaller instance of P_2.
3. If the solution to P_3 is known because it is small enough to be solved, then P_2 could be solved, and then the original problem P_1 could be solved.

Note : Do not confuse 'iteration', which requires loops, with recursion.

- Many programming languages allow recursive procedures: procedures which call themselves.
- Recursively defined sequences occur frequently in practice.

A sequence of numbers is defined recursively if

- Requirement **B** (Base values)

  a finite set of values is specified.

- Requirement **R** (Remaining values)

  the remaining values of the sequence are defined by a recursive function f whose domain includes previous values of the sequence.

Example

Define the sequence $FACT$ by

(B) $FACT(0) = 1$

(R) $FACT(n) = n \times FACT(n - 1)$ for $n \geq 1$.

This function defines the factorial of n, the product of all the natural numbers from 1 to n. Thus

1. $n = 0 : FACT(0) = 1.$
2. $n = 1 : FACT(1) = 1 \times FACT(0) = 1.$
3. $n = 2 : FACT(2) = 2 \times FACT(1) = 2.$
4. $n = 3 : FACT(3) = 3 \times FACT(2) = 6.$
5. $n = 4 : FACT(4) = 4 \times FACT(3) = 24.$

The following recursive procedure implements the factorial function.

```
type nonneg = 0 ... maxint;
function Fact ($n$ : nonneg) : nonneg;
Precondition : $n$ must be greater than or equal to 0.
Postcondition : Returns the factorial of $n$; $n$ is unchanged.
begin
    if $n = 0$
        then Fact := 1
        else Fact := $n *$ Fact($n - 1$)
end {Fact}
```

Example
The Fibonacci sequence is defined by
(B) $FIB(0) = FIB(1) = 1$
(R) $FIB(n) = FIB(n-1) + FIB(n-2)$ for $n \geq 2$.

1. $n = 0$: $FIB(0) = 1$.
2. $n = 1$: $FIB(1) = 1$.
3. $n = 2$: $FIB(2) = FIB(1) + FIB(0) = 2$.
4. $n = 3$: $FIB(3) = FIB(2) + FIB(1) = 3$.
5. $n = 4$: $FIB(4) = FIB(3) + FIB(2) = 5$.
6. $n = 5$: $FIB(5) = FIB(4) + FIB(3) = 8$.

The result is $1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ □

The following recursive procedure implements the Fibonacci function.

```
type nonneg = 0 ... maxint;
function Fib (n : nonneg) : nonneg;
Precondition : n must be greater than or equal to 0.
Postcondition : Returns the n-th Fibonacci number; n is unchanged.
begin
    if n ≤ 1
        then Fib := 1
        else Fib := Fib(n − 1) + Fib(n − 2)
end {Fib}
```

Example A binary search for a word in a dictionary.

```
Search (dictionary, word)
    if dictionary is one page in size
        then scan the page for the word
        else begin
            Open dictionary near the middle
            Determine the half of the dictionary in which the word is located
            if word is in the first half
                then Search (first half of dictionary, word)
                else Search (second half of dictionary, word)
end
```

Notes :

- The procedure calls itself, that is, the procedure *Search* calls *Search*, and thus it is recursive.
- Each call to *Search* is made with a dictionary whose size is one–half of that used in the previous call. Thus the procedure solves the search problem by solving another problem that is identical in nature but smaller in size.
- There is one search problem, the base case, that is handled differently to all the others. When *dictionary* contains a single page, it is scanned directly and the recursion stops.
- The manner in which the size of the problem diminishes guarantees that the base case will always be reached.

Example Writing a string of characters backwards

```
procedure WriteBackward (S : string; n : integer);
{The procedure writes the first n characters of S backwards.}
begin
    if n > 0
        then begin
            Write (S[n]); {write the last character}
            WriteBackward (S, n − 1) {write the rest of the string backward}
        end
        else {no action : n = 0, the base case}
end {WriteBackward}
```

Example
A recursive function that calculates the number of digits in a natural number n is :

$$\begin{array}{lcll} \texttt{digits}(n) & = & 1 & \text{if } n < 10 \\ \texttt{digits}(n) & = & 1 + \texttt{digits}(n \text{ div } 10) & \text{otherwise} \end{array}$$

The number of digits in 470 is

$$\begin{array}{lcl} \texttt{digits}(470) & = & 1 + \texttt{digits}(47) \\ & = & 1 + (1 + \texttt{digits}(4)) \\ & = & 1 + (1 + 1) \\ & = & 3 \end{array}$$

Induction and recursion

Inductive proofs and recursive definitions/procedures are closely related :

1. Base values
   - In proof by induction, the proposition is proved first for a specific value of n.
   - In definition by recursion, a finite number of values of the sequence are given.
2. Remaining values
   - In proof by induction, we assume that the proposition is true for the value n, and prove that it is true for $n+1$.
   - In (a simple) definition by recursion, the $(n+1)$th element of the sequence is defined in terms of the nth.

Typically, *induction* is required to prove statements about *recursive* programs.

Example
Consider the sum S_n of the first n positive numbers.

- Induction is used to prove that $S_n = \frac{n(n+1)}{2}$ and S_1 is assumed to be known. This generates the sequence

$$S = \{S_1, S_2, S_3, S_4, \dots\}.$$

- A recursive definition of the sequence S is

$$S_n = S_{n-1} + n.$$

  with base, where the recursion stops, $S_1 = 1$. This recursive definition may be used to prove, by induction, that the sequence so defined is $S_n = \frac{n(n+1)}{2}$.

The principle of mathematical induction (general).

Let $P(n)$ be a **proposition** involving the positive integer n.
If we know

1. $P(1)$ is true;
2. if $n > 1$, and $P(k)$ holds for all $1 \le k < n$, then $P(n)$;

then $P(n)$ is true for all $n \ge 1$.

The number 1 here can be replaced throughout by 0, or by any integer.

Proof of general induction

Let $Q(n)$ be a the proposition '$P(k)$ for all $k < n$'. Then we prove $Q(n)$ for all $n \ge 2$, by induction:

1. $Q(2)$ is true, since $P(1)$ is true;
2. if $Q(n)$ holds, then $P(k)$ holds for all $k < n$, so $P(n)$ holds, so $P(k)$ holds for all $k < n+1$, so $Q(n+1)$ holds;

Conclusion $Q(n)$ is true for all $n \ge 2$, by ordinary Mathematical Induction.

The steps involved in using general induction are :

1. The **induction base**: prove the statement for $n = 1$ (or whatever your starting value is)
2. The **induction step**: assuming the result for all numbers less than some arbitrary $n > 1$, (the **induction hypothesis**) deduce the same result for n.
3. **Conclusion** 'By mathematical induction, the result holds for all $n \ge 1$'.

Example Consider the Fibonacci sequence (f_n) defined recursively by $f_n = f_{n-1} + f_{n-2}$ $(n > 2)$ and $f_1 = f_2 = 1$. Use induction to prove that for all $n \ge 1$,

$$f_n < 2^n \qquad (n \ge 1).$$

Proof
Induction Base The induction base consists of the **two** cases: $n = 1$ and $n = 2$. (Alternatively, you may say that we prove the $n = 1$ case and then apply general induction starting at $n = 2$.) We observe that

$$f_1 = 1 < 2^1, \qquad f_2 = 1 < 2^2,$$

so the proposition is true for $n = 1$ and $n = 2$.

Induction Step The **Induction Hypothesis** is that

$$f_k < 2^k \qquad (1 \le k < n).$$

Assuming this we have, for $n > 2$,

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \\ &< 2^{n-1} + 2^{n-2}, \qquad \text{by the Induction Hypothesis,} \\ &= \frac{1}{2}2^n + \frac{1}{4}2^n \\ &= \frac{3}{4}2^n \\ &< 2^n. \end{aligned}$$

So the result for n follows.
Conclusion By mathematical induction, the result holds for all $n \ge 1$.

Propositional Logic

Propositional calculus is

- a model for simple reasoning – its original purpose
- hence vital for reasoning by computer, eg, theorem provers, program verifiers
- the basis of digital circuit design (on/off corresponding to true/false)

Propositional calculus is the study of logical relationships between objects called propositions which can be either true or false.

Example
Consider the statement
`if` $(a < b \vee (a \not< b \wedge c = d))\ldots$
Note :

1. $\vee$ denotes `OR`. A program would evaluate the second expression only if the first expression is FALSE
2. $\wedge$ denotes `AND`. Evaluate the second expression only if the first expression is TRUE

The expression simplifies to
`if` $(a < b \vee c = d)\ldots$

Properties of propositions :

- A proposition must assert a fact.
- Questions and commands are not propositions.
- Every proposition is either *true* or *false.*

Example

1. elephants are mammals
2. some birds can fly
3. France is in Asia
4. go away
5. are you happy ?
6. x is an even number

(1) , (2) and (3) are propositions,
(4) and (5) are not propositions.
(6) is a *predicate*, i.e. a proposition which depends on a variable.

What are the limitations of propositional logic ?

Consider the expression
`if` $(a < b \wedge a < c \wedge b < c)\ldots$
This is equivalent to
`if` $(a < b \wedge b < c)\ldots$
but it cannot be deduced by propositional logic because it contains predicates that may be true or false, depending on the values assigned to the arguments.
The above equivalence depends on a relation between the truth values of these predicates $L(x, y) \equiv (x < y)$ for different values of the arguments.

Composite statements and logical connectives

Definitions

- Compound statements :

  A *compound statement* is a statement that is composed of two or more propositions.

- Logical connectives :

  A *logical connective* combines two propositions in a compound statement.

Example
Paris is in England **and** $2 + 3 = 5$.
It rained in Sheffield on June 6, 1657 **and** 10 is greater than 9. $\square$

The logical connectives that are used to combine propositions include

and (conjunction)	$\wedge$
or (disjunction)	$\vee$
implication	$\Rightarrow$
not (negation)	$\neg$
equivalence	$\Leftrightarrow$

These logical connectives are defined by specifying how the truth or falsity of the compound statement depends on the truth or falsity of the two propositions being combined. This is best done by a *truth table.*

1. Conjunction : Two statements can be combined with the word *and* to form a compound statement of the original statements.
If p and q are two statements, their conjunction is $p \wedge q$.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

$p \wedge q$ is true if and only if p is true and q is true.

2. Disjunction : Two statements can be combined with the word *or* to form a compound statement of the original statements. This is 'or' in the inclusive sense: one or the other, *or both.* (Latin *vel* rather than *aut.*)

If p and q are two statements, their disjunction is $p \vee q$.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

$p \vee q$ is false if and only if p is false and q is false.

3. Implication : If p and q are two statements, their conditional implication is $p \Rightarrow q$.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

$p \Rightarrow q$ is true if and only if p and q are both true, or p is false, independent of the truth or falsity of q.

Thus a false proposition 'implies' anything because the conditional implication is defined to be true if p is false. Also any true statement 'implies' any other, whether or not their meanings are related.

We are not concerned with the inner meanings of propositions, just with their truth or falsity. This 'implication' in a technical sense, rather than in the ordinary sense of 'inference'.

Consider the statements :

- *If it rains, I carry my umbrella.*

  If it is not raining, I may or may not be carrying my umbrella.

- *If the power indicator light is lit, then the unit is connected to the supply.*

  Perhaps light is lit, in which case then the unit must be connected to the supply.

  If the light is off; then perhaps the unit is still connected to the supply, but the light is faulty.

4. Negation : If p is a statement, the negation of p is $\neg p$.

p	$\neg p$
T	F
F	T

5. Equivalence : The equivalence connective, (also called the biconditional connective,) $p \Leftrightarrow q$ is defined to be $(p \Rightarrow q) \wedge (q \Rightarrow p)$.

p	q	$p \Rightarrow q$	$q \Rightarrow p$	$p \Rightarrow q \wedge q \Rightarrow p$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

Thus the biconditional is defined by the following truth table.

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

This proposition is expressed linguistically by using the phrase *if and only if*, (iff).

Example I'll give you £10 if and only if you clean the car.

The 'if' direction says 'You clean the car $\Rightarrow$ I'll give you £10'.

The 'only if' direction says 'I'll give you £10 $\Rightarrow$ you clean the car'.

Two statements are equivalent if and only if their truth tables are the same.

Example

Show that $p \Rightarrow q \equiv \neg (p \wedge \neg q)$.

p	q	$p \Rightarrow q$	$\neg q$	$p \wedge \neg q$	$\neg (p \wedge \neg q)$
T	T	T	F	F	T
T	F	F	T	T	F
F	T	T	F	F	T
F	F	T	T	F	T

Example

Consider the statement

It is false that roses are red and violets are blue

This statement can be written in the form $\neg(p \wedge q)$ where p is the proposition that roses are red, and q is the proposition that violets are blue. By logical equivalence

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

and thus the given statement is equivalent to

Roses are not red, or violets are not blue

Rules for finding equivalences

We can first remove the connectives $\Rightarrow$ and $\Leftrightarrow$ using

$$P \Leftrightarrow Q \;\equiv\; (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

and

$$P \Rightarrow Q \;\equiv\; Q \vee \neg P.$$

Then we use the following rules

Commutative laws :

$$P \vee Q \equiv Q \vee P, \qquad P \wedge Q \equiv Q \wedge P.$$

- Associative laws :

$$\begin{aligned}(P \vee Q) \vee R &\equiv P \vee (Q \vee R)\\ (P \wedge Q) \wedge R &\equiv P \wedge (Q \wedge R)\end{aligned}$$

- Distributive laws :

$$\begin{aligned}P \vee (Q \wedge R) &\equiv (P \vee Q) \wedge (P \vee R)\\ P \wedge (Q \vee R) &\equiv (P \wedge Q) \vee (P \wedge R)\end{aligned}$$

- De Morgan's laws :

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q \qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q.$$

The validity of these equivalences can be checked by truth tables.

Notice that these are the same as the laws for set algebra, essentially because

$$x \in A \cap B \;\equiv\; x \in A \wedge x \in B,$$

et cetera. They are the laws of *Boolean algebra.*

Tautologies and Contradictions

A *tautology* is a logical expression that is always true, independent of the value of its variables, p, q, etc.

Example

p	q	$p \Rightarrow q$	$[p \wedge (p \Rightarrow q)]$	$[p \wedge (p \Rightarrow q)] \Rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Thus $[p \wedge (p \Rightarrow q)] \Rightarrow q$ is a tautology. $\square$

The opposite of a tautology is a *contradiction*: a logical expression that is always false, independent of the value of its variables, p, q, etc.

Example

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

Thus $p \wedge \neg p$ is a contradiction. $\square$

The Theorems of Propositional Logic.

Essentially, the theorems are precisely the tautologies.

For example, $P \Rightarrow P$ is a tautology, so it is a theorem and we write $\vdash P \Rightarrow P$.

To check whether a given statement is a theorem, draw up a truth table. (This could be tedious for complicated propositions involving many variables.)

Truth tables suffer from two disadvantages :

- They are unmanageable for a large number of different propositions.
- They cannot be used with predicate formulae.

Under these circumstances, it is necessary to consider an alternative method of obtaining logical conclusions from given 'initial conditions'. It must be possible to apply this method using a computer in order to provide a mechanical method of proving properties of specifications or models of complex systems.

This method, the axiomatic method, is based on axioms, substitutions and rules of inference.

The axiomatic method

In this approach we single out a small set of tautologies as *axioms*, from which the theorems will be deduced in a sequence of steps. Each step comprises either

(a) writing down and axiom , or

(b) writing down a earlier line with a substitution, or

(c) using a *Rule of Inference* applied to one or more earlier lines.

The axioms (Kleene 1952)

$$A \Rightarrow (B \Rightarrow A),$$

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)),$$

$$(A \wedge B) \Rightarrow A,$$

$$(A \wedge B) \Rightarrow B,$$

$$A \Rightarrow (B \Rightarrow (A \wedge B)),$$

$$A \Rightarrow (A \vee B),$$
$$B \Rightarrow (A \vee B),$$
$$(A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow ((A \vee B) \Rightarrow C)),$$
$$(A \Rightarrow B) \Rightarrow ((A \Rightarrow \neg B) \Rightarrow \neg A),$$
$$\neg\neg A \Rightarrow A.$$

These are the axioms for the $\Rightarrow$, $\wedge$, $\vee$ and $\neg$. The biconditional implication $\Leftrightarrow$ is assumed to be defined by

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A).$$

Substitution

If a compound proposition P is a tautology and if all occurrences of a variable p of P are replaced by a proposition E, then the resulting proposition P^* is also a tautology.

Example

It has been shown that $[p \wedge (p \Rightarrow q)] \Rightarrow q$ is a tautology.

1. If every occurrence of p is replaced by $E = q \Rightarrow r$, the tautology

$$[(q \Rightarrow r) \wedge ((q \Rightarrow r) \Rightarrow q)] \Rightarrow q$$

is obtained.

2. If, instead, every occurrence of q is replaced by E, the tautology

$$[p \wedge (p \Rightarrow (q \Rightarrow r))] \Rightarrow (q \Rightarrow r)$$

is obtained.

Rules of Inference

An inference rule is a list of propositions above a horizontal line and one proposition below the line. If a proof involves an expression that contains the formula above the line, then it can be replaced by the formula below the line.

There is just one rule of inference in propositional logic

Modus Ponens :

$$\frac{P \Rightarrow Q, \quad P}{Q}$$

from $P \Rightarrow Q$ and P we can deduce Q. (Substitutions may be made for both P and$Q$.)

Example

If $P \Rightarrow Q$ is true, we can deduce $\neg Q \Rightarrow \neg P$ because
$(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$ is a tautology.

We can deduce other rules of inference:

Example

$$\frac{\neg\neg A}{A} \qquad [DN]$$

states that if it has been established that $\neg\neg P$ is true, then I can deduce that P is also true. (DN denotes double negative.)

To prove this, we start with

$$\neg\neg A,$$

then write down the axiom

$$\neg\neg A \Rightarrow A,$$

then use Modus Ponens with $\neg\neg A$ in place of P, and A in place of Q.

$$\frac{\neg\neg A \Rightarrow A, \quad \neg\neg A}{A}.$$

Example

Elimination rules for conjunction

$$\frac{P \wedge Q}{P} \quad [\wedge E1], \qquad \frac{P \wedge Q}{Q} \quad [\wedge E2].$$

These rules state that if we know that $P \wedge Q$ is true, then P and Q are both true.

$E1$ and $E2$ denote that these are elimination rules.

Truth table form : $(P \wedge Q) \Rightarrow P$ is a tautology.

Example

Introduction rule for conjunction

$$\frac{P, \quad Q}{P \wedge Q} \quad [\wedge \mathrm{I}].$$

This rule states that if we know that P is true and P is true, then $P \wedge Q$ is true. Thus

I denotes that this is an introduction rule.

Example

Introduction rules for disjunction

$$\frac{P}{P \vee Q} \quad [\vee I1], \qquad \frac{Q}{P \vee Q} \quad [\vee I2].$$

These rules state that if we know that P is true or Q is true, then $P \vee Q$ is true.

$I1$ and $I2$ denote that these are introduction rules.

Elimination rule for disjunction

$$\frac{P \vee Q, \quad P \Rightarrow R, \quad Q \Rightarrow R}{R} \qquad [\vee E]$$

The elimination rule $\vee E$ states that if R follows from P and Q, then if one or both of P and Q is true, then R is true. This rule follows from the tautology

$$(P \vee Q) \wedge (P \Rightarrow R) \wedge (Q \Rightarrow R) \Rightarrow R.$$

The De Morgan rules

$$\frac{p \vee q}{\neg(\neg p \wedge \neg q)} \qquad [\vee\text{DM}]$$

$$\frac{p \wedge q}{\neg(\neg p \vee \neg q)} \qquad [\wedge\text{DM}]$$

These follow from the De Morgan laws.

Finite Automata

Many systems, including a computer, are defined by four terms :

- **Inputs** – the set of inputs from the external environment
- **States** – the set of suitable states of the system
- **Outputs** – the set of possible system outputs to the environment
- **Rules** – the laws that relate the future states/outputs to the current inputs/states/outputs

Informally, a state is a possible internal configuration of the system.

Informally:
A (deterministic) finite automaton is a machine which at any given time is in one of a certain **finite set Q of states**.

It has an input wire, through which it can be fed characters selected from its **input alphabet** Σ.

Each time a character is fed in, the machine changes to a new state according to a **transition function** δ. If it is in state q and is fed with a character a, the machine changes to state $\delta(q, a)$.

The only output consists of a light on the top of the machine which is on when the machine is in one the **acceptance states**. (This is a very rudimentary output mechanism — we shall consider more articulate machines later.)

The machine is started in the **initial state** q_0.

If, after inputting a string of characters $a_1a_2 \dots a_n$, the light is on, we say the string $a_1a_2 \dots a_n$ has been **accepted**. Thus, each automaton M is associated with a **set of strings** (i.e. a **language**) $L(M)$, namely the set of all strings which it accepts. We shall study this relationship between automata and languages, together with similar relationships for more powerful machines.

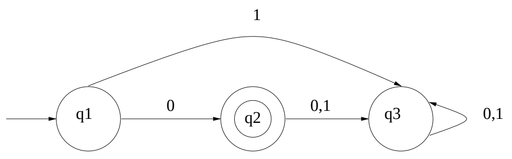
Some authors use the term 'finite state recognizer' instead of 'finite automaton'.

Formal Definition
A (deterministic) **finite automaton** (FA) is an ordered quintuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a **finite** set; the **set of states** — the machine is always in one of these states;
- Σ is a finite set of characters; the **input alphabet** — input strings are made up from these;
- $\delta : Q \times \Sigma \to Q$ is the **transition function** — if the machine is in state q and receives an input character a, it changes to state $\delta(q, a)$ and awaits the next input;
- $q_0 \in Q$; the **initial state** — the state in which the machine starts;
- $F \subseteq Q$; the set of **final** or acceptance states.

We can specify a FA's transition function either by giving a **transition table** or by a **state diagram**.

Example $\Sigma = \{0, 1\}$, $Q = \{q_1, q_2, q_3\}$, initial state $= q_1$, state diagram: This machine accepts the string

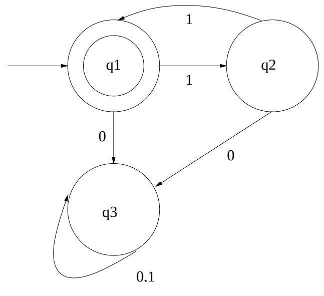


'0' and no other: $L(M) = \{0\}$.

Transition table

δ	0	1
q_1	q_2	q_3
q_2	q_3	q_3
q_3	q_3	q_3

Example $\Sigma = \{0, 1\}$, $Q = \{q_1, q_2, q_3\}$, initial state $= q_1$, $F = \{q_2\}$, state diagram as shown below. This machine accepts precisely those strings consisting of an even number (possibly zero) of 1s: $L(M) = \{\varepsilon, 11, 1111, 111111, \dots\}$.



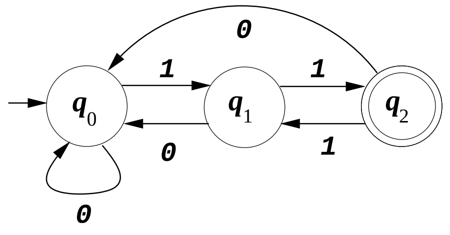
Transition table

δ	0	1
q_1	q_3	q_2
q_2	q_3	q_1
q_3	q_3	q_3

Example
$Q = \{q_0, q_1, q_2\}$; initial state q_0, $F = \{q_2\}$; and δ is given by the following transition table.

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_1

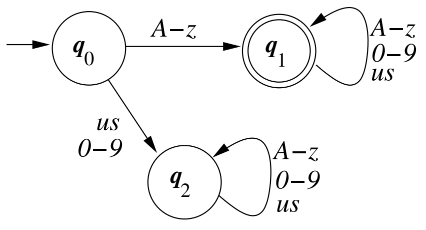
The state diagram is:



$L(M)$ is the set of all strings ending in a (non-zero) **even number of ones**.

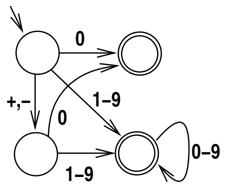
Example Variable/keywords names in C++
'C++ naming rules:
* The only characters you can use in names are alphabetic characters, digits, and the underscore (_) character.
* The first character in a name cannot be a digit.
* Uppercase characters are considered distinct from lowercase characters.
* You can't use a C++ keyword for a name.
* Names beginning with two underscore characters or with an underscore character followed by an uppercase letter are reserved for use by the implementation. Names beginning with a single underscore character are reserved for use as global identifiers by the implementation.
* C++ places no limits on the length of a name, and all characters in a name are significant.'

For simplicity, let us ignore the problem of avoiding C++ keywords and build a FA to recognize variable names and keywords, rejecting words beginning with an underscore. We assume that the input alphabet consists of just the letters of the alphabet in upper and lower case (A–z), the digits (0–9) and the underscore character (us).



Notice that q_2 acts as a 'rubbish state' which is not final and from which all transitions go to itself.

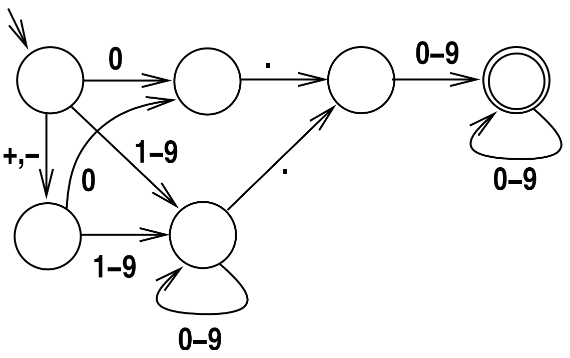
Example Integers in C++
An **integer** is a string of digits, possibly preceded by + or -, not beginning with 0 unless the string is just 0. (e.g. `1234, +1234, -1234, 0, +0, -0,` but not `+01234` (leading zero)). A (partial) state diagram of a machine recognizing integers is shown below. All



transitions not shown on the diagram are to a 'rubbish state' which is not final and from which all transitions go to itself.

Example Real numbers in C++
These can be written in two ways:
Fixed-point notation: i.e. (integer)`.`(non-empty string of digits) (e.g. `12.34, -12.034, +12.000,` but not `12.` or `1234`)
Floating-point or Scientific notation: (fixed point number or integer)(`e` or `E`)(integer) (e.g. `12.3e4, 1.234E0, 0.12e-34`)
A state diagram of a FA recognizing fixed-point numbers is shown below.
The problem of finding a FA to recognize floating-point numbers will be set for homework.

A machine recognizing fixed-point numbers



Again, transitions not shown on the diagram are to a 'rubbish state'.

Notation We write Σ^* for the set of all (empty or non-empty) finite strings of characters from Σ. We use ε to denote the **empty string** (some authors use λ). A **language** is a subset of Σ^*.

If $x, y \in \Sigma^*$, say $x = a_1a_2 \dots a_n$ and $y = b_1b_2 \dots b_m$, then we write xy for the string

$$xy = a_1a_2 \dots a_nb_1b_2 \dots b_m,$$

the **concatenation** of x and y.

Definition For a FA, $M = (Q, \Sigma, \delta, q_0, F)$, we define $\hat{\delta} : Q \times \Sigma^* \to Q$ recursively as follows.
(i) $\hat{\delta}(q, \varepsilon) = q$;
(ii) for $w \in \Sigma^*$ and $a \in \Sigma$,

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a).$$

[Informally, $\hat{\delta}(q, a_1a_2 \dots a_n)$ is the state you end in if you start in state q and input $a_1a_2 \dots a_n$.]

We can then formally define:
a string $w \in \Sigma^*$ is **accepted by** M if and only if $\hat{\delta}(q_0, w) \in F$;
the **language accepted by** M is

$$L(M) = \{w \in \Sigma^* : \ \hat{\delta}(q_0, w) \in F\}.$$

Remarks
(a) $\hat{\delta}(q, a) = \delta(q, a)$ for $a \in \Sigma$ — some authors drop the hat because of this.
(b) $\varepsilon \in L(M)$ iff $q_0 \in F$.

Theorem

$$\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y) \qquad (x, y \in \Sigma^*).$$

Proof We prove this by induction on $n = |y|$, the length of y.

Induction base. If $n = 0$, then $y = \varepsilon$ and $xy = x$, so

$$\hat{\delta}(\hat{\delta}(q, x), y) = \hat{\delta}(\hat{\delta}(q, x), \varepsilon) = \hat{\delta}(q, x).$$

Induction step. We now assume that the theorem is true for a given arbitrary value of n, and show that it is true for $n + 1$. Thus we assume:

Induction hypothesis:

$$\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y) \qquad (x \in \Sigma^*)$$

when $|y| = n$.

Suppose y' is a string of length $n+1$. Then $y' = ya$ for some string y of length n, and some character a. Then

$$\begin{array}{rcll} \hat{\delta}(q, xy') & = & \hat{\delta}(q, xya) & \\ & = & \delta(\hat{\delta}(q, xy), a), & \text{definition of } \hat{\delta}, \\ & = & \delta(\hat{\delta}(\hat{\delta}(q, x), y), a), & \text{Induction Hypothesis}, \\ & = & \hat{\delta}(\hat{\delta}(q, x), ya), & \text{definition of } \hat{\delta}, \\ & = & \hat{\delta}(\hat{\delta}(q, x), y'). & \end{array}$$

So the result holds for a string of length $n + 1$

Conclusion: By mathematical induction, the result holds for all $n \geq 0$, and the theorem is proven.

Languages

We study **formal languages**, such as programming languages constructed according to rigid rules, but these are influenced by studies of **natural languages** — English, French, Latin, Tolomako, ...– particularly by Noam Chomsky (see 'Syntactic Structures' (1957), Main Library 415.84 (C)). He, and others, attempt to fit syntactic structures to existing languages (but not Tolomako). We only have to deal with artificial languages constructed using those structures.

Specialists in artificial intelligence involving natural language recognition will need to study linguistics further.

Alphabets and Languages

In natural languages, the basic building block is the **word**, (or the 'morpheme', as in 'cats' = 'cat' + [plural]), and linguists are concerned with the rules by which these are built up into **sentences**. The 'alphabet' is the set of words (or morphemes): the **language** is the set of all those strings of words which are grammatical sentences.

In programming languages, like Pascal, the 'alphabet' consists of the keywords, variables, arithmetic operators, brackets,... and these are built up into **programs**. The language Pascal may be identified with the set of all strings of these lexical units which are **syntactically correct** programs (whether they work or not!).

To model these ideas, we simplify things and consider an **alphabet** Σ which is a set of symbols and a **language** L over Σ which is a **set of strings** of symbols in Σ. We write $L \subseteq \Sigma^*$.

Example If $\Sigma = \{a, b\}$, then
$\Sigma^* = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, aba, abb, baa, \dots\}$
A few languages over Σ are

1. $\{\varepsilon, a, aa, aab\}$
2. $\{x \in \Sigma^* : |x| \le 8\}$
3. $\{x \in \Sigma^* : |x| \text{ is odd }\}$
4. $\{x \in \Sigma^* : n_a(x) > n_b(x)\}$
5. $\{x \in \Sigma^* : |x| > 2 \text{ and } x \text{ begins and ends with } b\}$

$|x|$ denotes the number of symbols in the string x.
$n_a(x)$ and $n_b(x)$ denote the number of a's and b's in the string x.

The empty set and the null string

Note Languages are **sets of strings** of symbols.
$\emptyset$ is the empty **set** — a (very trivial) language containing no strings.
ε is the null **string**.
$\{\varepsilon\}$ is a **set** with one member — it's the (slightly less trivial) language which contains only one string, namely ε.

A helpful image of a language might be a ring-binder containing sheets of paper each with a string written on it:
$\{\varepsilon\}$ is a ring binder containing one sheet of blank paper;
$\emptyset$ is an empty ring binder!

Operations on languages

Since languages are sets of strings, new languages can be constructed by the set operations

- Union
- Concatenation
- Kleene closure

Question : Why are these operations important ?
Answer : If certain languages are recognized by finite automata, then the languages that result from the application of these operations *will also be recognized by finite automata*

Union operator
Let Σ be a finite set of symbols, and let $L_1, L_2 \subseteq \Sigma^*$ be two languages over Σ.
The *union* of L_1 and L_2, denoted by $L_1 \cup L_2$, is the set

$$\{x : x \in L_1 \text{ or } x \in L_2\}$$

which is the usual definition of the union of two sets.
Example With $\Sigma = \{0, 1\}$, if $L_1 = \{10, 1\}$ and $L_2 = \{011, 11\}$, then
$L_1 \cup L_2 = \{10, 1, 011, 11\}$

Concatenation operator
Let Σ be a finite set of symbols, and let $L_1, L_2 \subset \Sigma^*$ be two languages over Σ.
The *concatenation* of L_1 and L_2, denoted by L_1L_2, is the set

$$\{xy : x \in L_1 \text{ and } y \in L_2\}$$

which is the set of all possible combinations of a string from L_1 followed by a string from L_2.
Example
With $\Sigma = \{a, \dots, z\}$, if $L_1 = \{hope, fear\}$ and $L_2 = \{less, fully\}$, then
$L_1L_2 = \{hopeless, hopefully, fearless, fearfully\}$

Kleene (Closure) operator
Use exponential notation to indicate the number of items being concatenated
If Σ is an alphabet and $a \in \Sigma$, $x \in \Sigma^*$, and $L \subseteq \Sigma^*$, then

1. $a^k = aa \cdots a$
2. $x^k = xx \cdots x$
3. $L^k = LL \cdots L$

where there are k factors in each product.

If $k = 0$, then

$$a^0 = \varepsilon, \qquad x^0 = \varepsilon, \qquad L^0 = \{\varepsilon\}$$

L^k is the set of all strings that can be obtained by concatenating k elements of L.

Example If $L = \{a, b\}$ where a and b are strings, then

1. $L^0 = \{\varepsilon\}$ where ε is the empty string
2. $L^2 = LL = \{a, b\}\{a, b\} = \left\{a^2, ab, ba, b^2\right\}$
3. $$\begin{aligned} L^3 &= L^2L = \{a, b\}\left\{a^2, ab, ba, b^2\right\} \\ &= \left\{a^3, a^2b, aba, ab^2, ba^2, bab, b^2a, b^3\right\} \end{aligned}$$

Extend to include the set of all strings that can be obtained by concatenating any number of elements of L :

$$L^0 = \{\varepsilon\}, \qquad L^{i+1} = LL^i$$

The *Kleene closure* L^* of L is defined as

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Example If $L = \{a, b\}$ where a and b are strings, then

$$\begin{aligned} L^* = \{&\varepsilon, a, b, a^2, ab, ba, b^2, a^3, a^2b, aba, ab^2, ba^2, bab, \\ &b^2a, b^3, \cdots\} \end{aligned}$$

Example If $L = \{10, 1\}$, then $L^* = \{\varepsilon, 10, 1, 1010, 101, 110, 11, \cdots\}$

Fact
If L, L_1 and L_2 are two languages over Σ that are recognized by finite automata, then

1. $L_1 \cup L_2$,
2. L_1L_2 and
3. L^*

are also recognized by finite automata.
These operations provide a set of rules that allow us to build up formal languages recognized by finite automata.

Kleene's Theorem

The following languages over an alphabet Σ are recognized by finite automata:

- the empty language $\emptyset$;
- the language $\{\varepsilon\}$;
- for each $a \in \Sigma$, the language $\{a\}$;
- any language produced from these basic building blocks by the (repeated) application of the operations of union, concatenation and closure.

Kleene's Theorem. These and only these languages are accepted by finite automata.

Definition Let Σ be an alphabet; the **regular expressions** over Σ and the sets that they represent are defined recursively :

1. $\emptyset$ is a regular expression and represents the empty set $\{\}$
2. $\boldsymbol{\varepsilon}$ is a regular expression and represents the set $\{\varepsilon\}$, the set that only contains the string of zero length
3. For each $a \in \Sigma$, $\boldsymbol{a}$ is a regular expression and represents the set $\{a\}$
4. If $\boldsymbol{R}$ and $\boldsymbol{S}$ are regular expressions denoting the sets R and S respectively, then $\boldsymbol{R} \cup \boldsymbol{S}$ (union), $\boldsymbol{RS}$ (concatenation) and $\boldsymbol{R}^*$ (Kleene closure) are regular expressions that represent the sets $R \cup S$, RS and R^* respectively

A string is a regular expression if and only if it can be derived from the regular expressions (1), (2) and (3) by a finite number of applications of the rules (4)

Regular expressions and regular languages

Notice that regular **expressions** are simply algebraic expressions. Each regular expression corresponds to a set of strings (i.e. a language)

The languages corresponding to regular expressions are called **regular languages**.

Kleene's Theorem says that the languages accepted by finite automata are just the regular languages.

Not all languages are regular

Example The language

$$\{0^n1^n : n = 1, 2, 3, \dots\}$$

is not regular.

To recognise this language, a finite automaton would have to keep track of how many zeros had been input, up to an indefinitely large number, and you cannot store an indefinitely large number in a machine with only a finite number of states. (The number of states is fixed before you know how long the input might be.)

Examples Let Σ be the alphabet $\{0, 1\}$.

- $\mathbf{00}$ is a regular expression representing the set $\{00\}$
- $\mathbf{0}\cup\mathbf{1}$ represents $\{0, 1\}$
- $\mathbf{0}^*$ represents $\{\varepsilon, 0, 00, 000, \cdots\}$
- $(\mathbf{0}\cup\mathbf{1})^*$ represents the set of all strings of 0s or 1s, including the empty string
- $(\mathbf{0}\cup\mathbf{1})^*\mathbf{00}(\mathbf{0}\cup\mathbf{1})^*$ represents all strings with at least two consecutive 0s

Example Let Σ be the alphabet $\{0, 1\}$. The string

$$\mathbf{1}^*\mathbf{0}(\mathbf{01})^*$$

is a regular expression representing the set of all strings consisting of any number (including none) of 1s, followed by a single 0, followed by any number (including none) of 01 pairs.

Example Let Σ be the alphabet $\{0, 1\}$. The string

$$\mathbf{0} \cup \mathbf{00} \cup \mathbf{000} \cup \mathbf{1}^*$$

is a regular expression representing the set of all strings consisting of no more than three 0s, or of any number (including none) of 1s.

Example Let Σ be the alphabet $\{0, 1\}$. The string

$$\mathbf{11}((\mathbf{10})^*\mathbf{11})^*\mathbf{00}^*$$

is a regular expression representing the set of all strings consisting of a nonempty string of 11 pairs, interspersed with any number (including none) of 10 pairs, followed by at least one 0.

Example Let Σ be the alphabet $\{0, 1\}$. The string

$$(\mathbf{1}(\mathbf{11})^*\mathbf{0})^*\mathbf{1}(\mathbf{11})^*$$

is a regular expression representing the set of all strings consisting of...

The string

$$\mathbf{1}(\mathbf{11})^*$$

is a regular expression representing the set of all strings consisting of a single 1 followed by zero or more pairs of 1s; i.e. strings consisting of an odd number of 1s.

Hence, the string

$$(\mathbf{1}(\mathbf{11})^*\mathbf{0})^*\mathbf{1}(\mathbf{11})^*$$

is a regular expression representing the set of all strings consisting of an odd number of 1s, followed by a 0, followed by an odd number of 1s, followed by a 0, followed by an odd number of 1s, etc., ending in an odd number of 1s, the total number of interspersed 0s being zero or more.

Examples

1. The string 10100010 is in the set represented by the regular expression
$$(\mathbf{0}^*\mathbf{10})^*.$$
2. The string 011100 is NOT in the set represented by the regular expression
$$(\mathbf{0} \cup (\mathbf{11})^*)^*.$$
3. The string 000111100 is in the set represented by the regular expression
$$((\mathbf{011} \cup \mathbf{11})^*(\mathbf{00})^*)^*.$$

Example Find a regular expression over $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ for the set of all non-negative real numbers in fixed point notation.

To save writing, we set

$$\boldsymbol{N} = \mathbf{1} \cup \mathbf{2} \cup \mathbf{3} \cup \mathbf{4} \cup \mathbf{5} \cup \mathbf{6} \cup \mathbf{7} \cup \mathbf{8} \cup \mathbf{9}$$

$$\boldsymbol{D} = \mathbf{0} \cup \boldsymbol{N},$$

so these correspond to the sets of strings
$N = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
Then the regular expression

$$(\boldsymbol{ND}^* \cup \mathbf{0})\,\boldsymbol{.DD}^*$$

corresponds to the set of all non-negative real numbers.

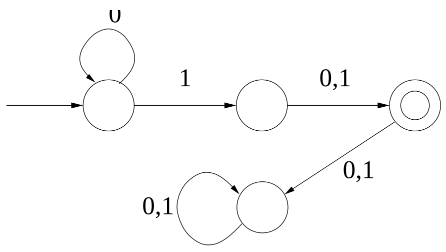
Note : A regular set may be described by more than one regular expression :
Example The set of all strings of 0s and 1s can be in represented in (at least) two ways :

$$(\mathbf{0} \cup \mathbf{1})^* = [(\mathbf{0} \cup \mathbf{11}^*)^* \cup (\mathbf{01})^*]^*$$

An efficient algorithm that enables one to determine if two regular sets are equal has not yet been established.

Examples of finite automata for regular expressions

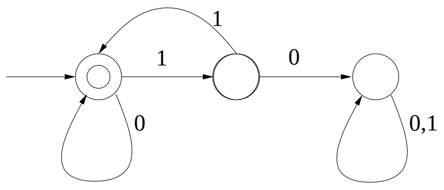
Example 1 The finite automaton corresponds to



the regular expression

$$\mathbf{0}^*\mathbf{1}(\mathbf{0} \cup \mathbf{1}).$$

Example 2 The finite automaton corresponds to



the regular expression

$$(\mathbf{0} \cup (\mathbf{11})^*)^* = (\mathbf{0} \cup (\mathbf{11}))^*.$$

Regular Expressions in Action I

Lexical Analysis
The **tokens** of a programming language – identifiers, literals, operators, reserved words and punctuation – can be described by a regular language.
During *lexical analysis* – the first stage of compilation – a software version of the corresponding finite automaton is used to recognisee these tokens.
(Of course, it is actually a finite automaton with output, but recognition is the key to its operation.)

Regular Expressions in Action II

Regular expressions are the standard method for characterizing sets of text sequences in

- web search engines
- UNIX: vi, Perl, Emac, egrep
- Windows editors: WORD, WinEdt, TextPad, MultiEdit

However, these implementations use a wide variety of different notations, always include operators other than the basic three, and frequently do not implement the full range of regular expressions.

egrep

`egrep` – extended global regular expression print – is a standard Unix tool that uses regular expressions to enable powerful searching. It searches a file for a specified pattern of characters and prints all lines that contain the pattern

Examples of `egrep`

- A string *s matches* a regular expression *r* if *s* is in the set of strings represented by *r*
- Given a regular expression, `egrep` will print every line of the file that contains a substring that matches the regular expression

Example
```
egrep 'depend' /usr/dict/words
depend
dependent
independent
```

The `egrep` notation

- Closure (*) is written as *
- Union (∪) is written as |
- Concatenation is denoted by adjacency

Example
$(\mathbf{0}{\cup}\mathbf{1})^*\mathbf{00}(\mathbf{0}{\cup}\mathbf{1})^*$ is denoted by (0|1)*00(0|1)*
The argument in the `egrep` command is a regular expression. By Kleene's theorem, a there is a finite automaton corresponding to this expression. The `egrep` program, in effect, builds a software model of this FA.

`egrep` syntax

- **Alphabet** The period character . matches any character in the alphabet Σ. The set Σ^* is matched by .*
- **Empty String** `egrep` cannot express the empty string ε, but r? denotes 0 or 1 occurrences of the regular expression r
- **Beginning of line** This is denoted by ˆ
- **End of line** This denoted by $

On a Solaris or Unix system, the file `/usr/dict/words` contains about 25 000 English words, one per line.

Example

- To find all words containing `depend` as a substring :
```
egrep 'depend' /usr/dict/words
depend
dependent
independent
```
- To find all words containing `depend` as a prefix :
```
egrep '^depend' /usr/dict/words
depend
dependent
```
- To find all words that start and end with `y`
```
egrep '^y.*y$' /usr/dict/words
yeasty
yeomanry
yesterday
```
- But ...
```
egrep 'y.*y' /usr/dict/words
alleyway
[...]
yesteryear
```

- Spell checking
```
egrep 'rec(ie|ei)ve' /usr/dict/words
receive
```
- Crossword puzzle : "s-blank-blank-u-blank-t-blank-blank-e"
```
egrep '^s..u.t..e$' /usr/dict/words
seductive
structure
```

Finite State Machines with Output

Example Pocket calculator
Input: buttons pressed.
Output: characters on LCD display
A finite machine but with a very large number of internal states. The following two examples have only modest numbers of states and could be implemented directly by electronic circuits.

Example Vending machine.
Input: coins, buttons pressed.
Output: instructions, coffee, tea, hot chocolate.

Example Traffic lights controller.
Input: signals from detectors and pedestrian push-buttons.
Output: different configurations of the lights.

Example Video recorder
Input: signals from the remote control.
Output: commands to start/stop motor, eject, change channel,. . . .

Often, machines will be implemented in software. The following are pure software examples
Example Compiler
Input: source code
Output: executable code
Finite machines are a poor model here, but a good model for the first stage (lexical analysis).

Example Language translator
Input: Russian text
Output: English text
Finite machines are a good model only for the most rudimentary translators.

Example Arithmetic unit: e.g. a binary adder
Input: two numbers, least significant digits (of both) first
Output: their sum

Mealy and Moore machines

Mealy or Moore machines are finite automata with the ability to write output characters.

- Mealy machine - an output is associated with each transition (as the machine receives an input and moves from one state to another state)
- Moore machine - an output is associated with each state, rather than with the transition between states

Definition of a Mealy machine :

A Mealy machine consists of a sextuple $M = (Q, \Sigma, \Gamma, q_0, \delta, \omega)$ where:

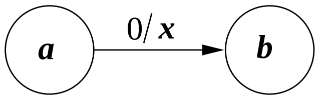
- Q is a finite set of states;
- Σ is a finite set of input symbols;
- Γ is a finite set of output symbols;
- q_0 is the initial state;
- $\delta : Q \times \Sigma \to Q$ is the state transition function;
- $\omega : Q \times \Sigma \to \Gamma$ is the output function.

If M is in state $q \in Q$ with an incoming input $a \in \Sigma$, the state changes to $\delta(q, a) \in Q$ and outputs the character $\omega(q, a) \in \Gamma$.

Example Let $Q = \{p, q, r\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{x, y, z\}$, initial state = p. We describe δ and ω by the following transition / output table

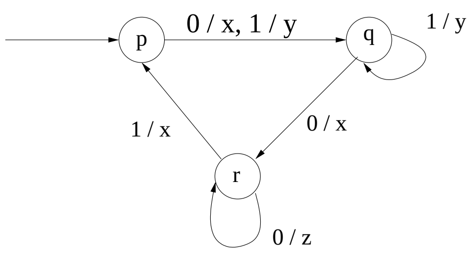
	input	p	q	r
δ	0	q	r	r
δ	1	q	q	p
ω	0	x	x	z
ω	1	y	y	x

In drawing state diagrams, we use the following notation.



The input 0 causes a state transition from a to b, and an output of x.

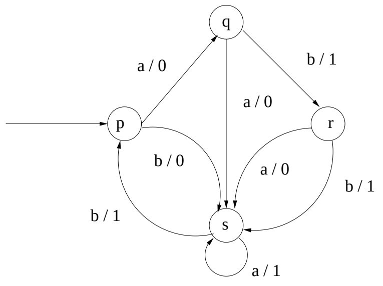
Example



Note: no 'final states'; Mealy machines have a better form of output.

Example

What is the output of the Mealy machine on input *ababb* ?



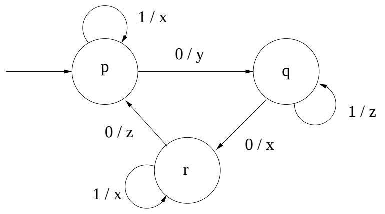
1. The first input letter is a, which takes us to q, and the output is 0.
2. The next input letter is b, which takes us to r, and the output is 1.
3. The next input letter is a, which leaves us at s, and the output is 0.
4. The next input letter is b, which takes us to p, and the output is 1.
5. The next input letter is b, which takes us to s, and the output is 0.

Thus the output is 01010.

Note : The length of the output string is equal to the length of the input string (=5).

Example

What is the output of the following machine on input 101 ?



Answer : xyz

Example Design a Mealy machine to add two numbers in binary form. The numbers will have to be fed in backwards: least significant digits first. They will also have to have sufficient leading zeros (fed in last) to make both numbers of the same length and so that each has at least one leading zero, to allow time for the machine to output a final carry digit.

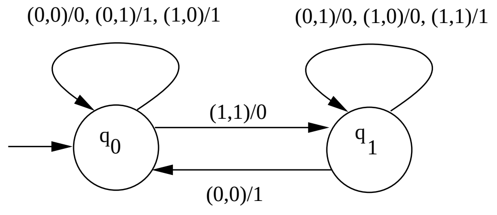
To deal with the two input streams, we simply have as the input alphabet

$$\Sigma = \{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}.$$

The output alphabet is $\Gamma = \{0,1\}$. There are just two states, q_0 and q_1, corresponding to the carry digit being 0 or 1. The initial state is q_0 and the transition / output table is the following.

	input	q_0	q_1
δ	$(0,0)$	q_0	q_0
δ	$(0,1)$	q_0	q_1
δ	$(1,0)$	q_0	q_1
δ	$(1,1)$	q_1	q_1
ω	$(0,0)$	0	1
ω	$(0,1)$	1	0
ω	$(1,0)$	1	0
ω	$(1,1)$	0	1

Here is the state diagram.



The function $\widehat{\omega}$

As with the transition function δ, so with the output function ω, we can extend the function to describe the behaviour when the machine is fed with a whole string rather than a single character. Thus, $\widehat{\omega}(q, a_1a_2\ldots a_n)$ will be the output string you get if you start in state q and input $a_1a_2\ldots a_n$.

We define $\widehat{\omega} : Q \times \Sigma^* \to \Gamma^*$ recursively as follows.
(i) $\widehat{\omega}(q,\varepsilon) = \varepsilon$;
(ii) for $w \in \Sigma^*$ and $a \in \Sigma$,

$$\widehat{\omega}(q, wa) = \widehat{\omega}(q,w)\omega(\hat{\delta}(q,w), a).$$

Definition Two Mealy machines

$$M = (Q,\Sigma,\Gamma,q_0,\delta,\omega), \qquad M' = (Q',\Sigma,\Gamma,q_0',\delta',\omega'),$$

with the same input alphabets and the same output alphabets, are **equivalent** if they produce the same output from any given input: formally, if

$$\widehat{\omega}(q_0,w) = \widehat{\omega'}(q_0',w) \qquad \text{for every } w \in \Sigma^*.$$

Some authors introduce a **translation function** $f_M : \Sigma^* \to \Gamma^*$ defined by

$$f_M(w) = \widehat{\omega}(q_0,w).$$

Then we can just say that M and M' are equivalent if and only if $f_M = f_{M'}$.

Minimization of Mealy Machines

Although we have treated finite state machines as abstractions, electronic devices (eg., logic elements) may be used to build circuits that act like finite state machines. In order to keep the number of internal states to a minimum, we try to 'minimize' a finite state machine.

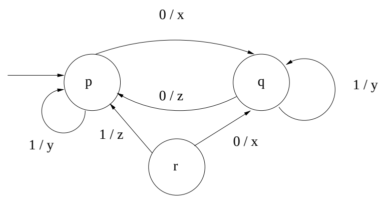
Let M be a given finite state machine. Minimization of M is the process of finding another machine M' with the following properties.

1. M' is equivalent to M.
2. M' has as few states as possible.

Two stages :

1. eliminate *unreachable* states;
2. merge *equivalent* states.

Stage 1 : Eliminating *unreachable* states.



The state r is unreachable.

Example
A Mealy machine is represented by the following table. Without drawing the state diagram, determine the states unreachable from the initial state p.

	input	p	q	r	s	t
δ	0	q	t	r	s	p
δ	1	t	q	s	q	p
ω	0	x	y	y	x	y
ω	1	x	y	y	x	y

1. We start in state p
2. Inputs 0,1 lead to states q, t respectively, so these are reachable.
3. From states q, t further inputs only lead to states p, q, t which are already on our list of reachable states.
4. Thus only states p, q and t are reachable. The states r and s are unreachable and can be deleted to produce a smaller equivalent machine.

Stage 2 : Merge *equivalent* states

Definition : Two states p and q in a Mealy machine are equivalent if they yield the same output for every given input, i.e. if

$$\widehat{\omega}(p, w) = \widehat{\omega}(q, w) \qquad \text{for all } w \in \Sigma^*.$$

The set of states is divided up into **equivalence classes**, so that two states are equivalent if and only if they belong to the same equivalence class. We then produce a machine M' whose states correspond to the equivalence classes of M.

The problem is: how do we tell when states are equivalent? It seems that we have to consider indefinitely long input strings w.

Definition Two states p, q of a Mealy machine M are ***k*-equivalent** if they yield the same output for every given input of length k, i.e. if

$$\widehat{\omega}(p, w) = \widehat{\omega}(q, w) \qquad \text{for all } w \in \Sigma^k.$$

For example, p, q are 1-equivalent if $\omega(p, a) = \omega(q, a)$ for all $a \in \Sigma$.

We can characterize k-equivalence recursively as follows:

(i) any two states are 0-equivalent;

(ii) two states p, q are k-equivalent if and only if

$$\omega(p, a) = \omega(q, a) \qquad \text{for all } a \in \Sigma$$

and

$\delta(p, a)$ and $\delta(q, a)$ are $(k-1)$-equivalent for all $a \in \Sigma$.

We start by determining which states are 1-equivalent; we divide up the set of states into **1-equivalence classes**.

We then use (ii) above to determine the 2-equivalent states. Note that 2-equivalence is a stronger condition than 1-equivalence: i.e. 2-equivalent states are always 1-equivalent. The converse is false: states may be 1-equivalent but not 2-equivalent. The result is that the 1-equivalence classes may break up into smaller 2-equivalence classes.

We proceed to 3-equivalence, 4-equivalence etc..

Eventually, we reach a point where k-equivalence is the same as $(k-1)$-equivalence. It follows from (ii) that $(k+1)$-equivalence will be the same as k-equivalence, and then $(k+2)$-equivalence will be the same, and so on.

At this point, we stop. These k-equivalence classes will be the desired equivalence classes

Example

Minimize the following Mealy machine, with start state p.

		p	q	r	s	t	u
δ	0	q	p	t	u	r	q
δ	1	r	t	s	r	s	r
ω	0	x	y	y	x	y	y
ω	1	y	x	y	y	y	x

1-equivalent states are those states that produce the same output for an input string of length 1 (a single character)

The 1-equivalence classes of M are $\{p, s\}, \{q, u\}$ and $\{r, t\}$

Finding the 2-equivalence classes : Consider each of the 1-equivalence classes separately.

- States p and s move to states q and u for an input of 0.
- States p and s move to state r for an input of 1.

Since q and u are 1-equivalent, and r is trivially 1-equivalent to itself, it follows that $\{p, s\}$ is a 2-equivalence class.

- States q and u move to states p and q for an input of 0.
- States q and u move to states t and r for an input of 1.

Since p and q are NOT 1-equivalent, q and u are not 2-equivalent.

1. States r and t move to states t and r for an input of 0.
2. States r and t move to state s for an input of 1.

Since t and r are 1-equivalent, and s is 1-equivalent to itself, it follows that $\{r, t\}$ is a 2-equivalence class.

Thus the 2-equivalence classes are

$$\{p, s\},\ \{q\},\ \{u\},\ \{r, t\}$$

Finding the 3-equivalence classes :

- States p and s move to states q and u for an input of 0. As q and u are not 2-equivalent, p and s are not 3-equivalent.
- r and t are 3-equivalent because r and t are 2-equivalent, and s is 2-equivalent to itself.

Thus the 3-equivalence classes are

$$\{p\},\ \{s\},\ \{q\},\ \{u\},\ \{r,t\}$$

Finding 4-equivalent states :
States r and t move to states t and r for an input of 0 and to state s for an input of 1. Since $\{r,t\}$ and $\{s\}$ are 3-equivalence classes, the 4-equivalence classes are the same as the 3-equivalence classes, and no further refinement is possible.
The equivalence classes are

$$\{p\},\quad \{s\},\quad \{q\},\quad \{u\},\quad \{r,t\}.$$

The new Mealy machine is obtained by defining a new state w corresponding to the only non-trivial equivalence class, $\{r,t\}$. You could say '$w = r = t$'.

	input	p	q	w	s	u
δ	0	q	p	w	u	q
δ	1	w	w	s	w	w
ω	0	x	y	y	x	y
ω	1	y	x	y	y	x

Example
Minimize the following Mealy machine, with start state p

	input	p	q	r	s	t	u	v
δ	0	r	s	p	q	v	r	t
δ	1	s	r	t	u	u	p	p
ω	0	x	y	x	y	y	x	y
ω	1	x	y	x	y	y	x	y

The 1-equivalence classes are
$\{p,r,u\}$ and $\{q,s,t,v\}$

Calculate the 2-equivalence classes. Consider each of the 1-equivalence classes separately.

- States p and r move to states r and p for an input of 0
- States p and r move to states s and t for an input of 1

Since r and p are 1-equivalent, and s and t are 1-equivalent, it follows that $\{p,r\}$ is a 2-equivalence class

- States p and u move to state r for an input of 0
- States p and u move to states s and p for an input of 1

Since s and p are NOT 1-equivalent, p and u are not 2-equivalent. Thus the 1-equivalence class $\{p,r,u\}$ splits into two 2-equivalence classes: $\{p,r\}$ and $\{u\}$.

Consider now the 1-equivalence class $\{q,s,t,v\}$.

- States q and s move to states s and q for an input of 0
- States q and s move to states r and u for an input of 1

Since s and q are 1-equivalent, and r and u are 1-equivalent, it follows that $\{q,s\}$ is a 2-equivalence class

- States q and v move to states s and t for an input of 0
- States q and v move to states r and p for an input of 1

Since s and t are 1-equivalent, and r and p are 1-equivalent, it follows that q and v are 2-equivalent.
Similarly, q and t are 2-equivalent, and thus $\{q,s,t,v\}$ is a 2-equivalence class.
Thus the 2-equivalence classes are $\{p,r\}$, $\{u\}$ and $\{q,s,t,v\}$

To calculate the 3-equivalence classes, consider the 2-equivalent states.

- p,r are 3-equivalent.
- u is 3-equivalent to itself.
- q,v are 3-equivalent.
- s,t are 3-equivalent.
- q,s are not 3-equivalent.

Thus the 3-equivalence classes are $\{p,r\}$, $\{u\}$, $\{q,v\}$ and $\{s,t\}$.
Note: to show that $\{q,v\}$ and $\{s,t\}$ are complete 3-equivalence classes, we only need the fact that some member of one class is not 3-equivalent to some member of the other: in our case, q is not 3-equivalent to s.
Exercise : Show that the 4-equivalence classes are the same as the 3-equivalence classes, and thus no further reduction is possible.

A minimal Mealy machine is defined by defining $a = p = r$, $b = q = v$ and $c = s = t$:

	input	a	b	c	u
δ	0	a	c	b	a
δ	1	c	a	u	a
ω	0	x	y	y	x
ω	1	x	y	y	x

Summary : Minimization algorithm

1. Remove unreachable states.
2. Compute equivalence classes by recursively computing the 1-, 2-, ..., k-equivalent states, until the k- and $(k+1)$- equivalence classes are identical.
3. Merge states in each equivalence class.

Remark
Let $p_1, p_2, \ldots, p_n$ be k-equivalent, but not necessarily equivalent, states. If these n states are merged into one, then the system will produce exactly the same output for all inputs of length less than or equal to k but the original machine and the reduced machine may produce different outputs for strings of length greater than k.

Moore Machines

Moore machines output when in a state rather than when moving from one state to another. They are equivalent in power to Mealy machines, but often require more states than the equivalent Mealy machines. However, in some situations they may provide more natural models.
Note that in Gersting 'Mathematical Structures for Computer Science' Moore machines are simply called 'finite state machines' and Mealy machines are not treated.

Definition A Moore machine is a sextuple $M = (Q, \Sigma, \Gamma, q_0, \delta, \omega)$ where:

- Q is a finite set of states;
- Σ is a finite set of input symbols;
- Γ is a finite set of output symbols;
- q_0 is the initial state;
- $\delta : Q \times \Sigma \to Q$ is the state transition function;
- $\omega : Q \to \Gamma$ is the output function.

When M is in state $q \in Q$ it outputs the character $\omega(q) \in \Gamma$.

Example A good example where a Moore machine provides a suitable model is a traffic lights controller. (See Carroll and Long 'Theory of Finite Automata' Example 7.16). Moore machines can be represented by state diagrams in which states are labelled 'q/x' where q is the state and x the character output in that state.

As with Mealy machines, one can define $\widehat{\omega}(q, w) \in \Gamma^*$, the result of starting in state q and inputting the string w. The definition is by recursion on the length of w, but building up w from the right:
(i) $\widehat{\omega}(q, \varepsilon) = \varepsilon$;
(ii) for $w \in \Sigma^*$ and $a \in \Sigma$,

$$\widehat{\omega}(q, aw) = \omega(\delta(q, a))\widehat{\omega}(\delta(q, a), w).$$

Note that $\omega(q)$ does not form part of $\widehat{\omega}(q, w)$. We reckon that the machine does not start outputting until after it has made its first move.

Again, one may introduce a **translation function** $f_M : \Sigma^* \to \Gamma^*$ defined by

$$f_M(w) = \widehat{\omega}(q_0, w)$$

and say that M andM' are equivalent if and only if $f_M = f_{M'}$.
Moore machines can be mimimized in a similar way to Mealy machines — we omit the details.

Linguistics

We seek a theory of grammar which generates all grammatical (syntactically correct) sentences of the language, and only these, rather in the way that a theory of chemical structure might generate all possible molecules and only those.
We distinguish between *syntax* and *semantics.*

> Colourless green ideas sleep furiously.
> Furiously sleep ideas green colourless.

Both are *semantically* incorrect — i.e. nonsense — but the first is recognized as syntactically correct.

The first will typically be read with normal sentence intonation: the second as a series of disconnected words, with falling intonation on each word.
The syntax/semantics distinction is less easy in computer languages; is an undeclared variable a syntactic or a semantic error? In formal definitions of the syntax of a language full declaration of variables

is often not included — but it would be checked for by the compiler rather than allowed to cause a run-time error (as in BASIC).

In discussing natural languages,

- the 'alphabet' is the set of all words (essentially);
- the 'language' is the set of all grammatically correct sentences.

In discussing programming languages,

- the 'alphabet' is the set of all lexical units — keywords, variables, numbers, separators;
- the 'language' is the set of all syntactically correct programs.

Is English a regular language? NO!

The following are grammatical sentences.

God is dead. (Nietzsche)
The man who said that God is dead is dead.
The man who said that the man who said that God is dead is dead is dead.
The man who said that the man who said that the man who said that God is dead is dead is dead is dead.
...

The number of occurrences of 'the man who said that' has to match the number of occurrences of 'is dead'. This cannot be checked by a finite automaton. c.f. the bracket-matching problem discussed in an earlier problem sheet.

Objection: all but the first few of these are so long as to be unwieldy and not recognizable as grammatical.

One could go on to say that only sentences of less than one million words are grammatical, so English is finite, so regular.

But: (i) It is difficult to see a sharp cut-off on grounds of length.

(ii) Although English, if finite, would be regular, the FA recognizing it might be nothing more than a device containing a list of all English sentences. This sort of theory gives us no insight at all. We want a grammar small compared to the language (in a vague sense). The best way to get this is to *simplify* English by assuming it is infinite.

Chomsky's Phrase Structure Grammars

Let

S= sentence,		
NP= noun phrase	e.g.	'the belligerent cat',
		'the cat which caught the mouse',
		'the heart of the matter',
VP= verb phrase	e.g.	'sat on the mat',
		'ate the fish',
		'slept',
N= noun	e.g.	'cat', 'mat', 'fish', 'camel'
V = verb	e.g.	'ate', 'sat', 'slept', 'run'
T = definite article		'the'

S, NP, VP, N, V, T are **non-terminal variables**.

We start with S and **elaborate** it according to a certain (finite) set of **productions**, which form the grammar, until we get a string of **terminal variables** — i.e. words (roughly).

Sample productions

$$\begin{aligned} S &\to NP + VP \\ NP &\to T + N \\ VP &\to V + NP \\ VP &\to V \\ N &\to \text{cat, fish, } \textit{et cetera} \\ V &\to \text{ate, slept, } \textit{et cetera} \\ T &\to \text{the} \end{aligned}$$

Example: generating a sentence.

$$\begin{aligned} S &\to NP + VP \\ &\Rightarrow T + N + VP \\ &\Rightarrow T + N + V + NP \\ &\Rightarrow \text{the} + N + V + NP \\ &\Rightarrow \text{the} + \text{cat} + V + NP \\ &\Rightarrow \text{the} + \text{cat} + \text{ate} + NP \\ &\Rightarrow \text{the} + \text{cat} + \text{ate} + T + N \\ &\Rightarrow \text{the} + \text{cat} + \text{ate} + \text{the} + N \\ &\Rightarrow \text{the} + \text{cat} + \text{ate} + \text{the} + \text{fish} \end{aligned}$$

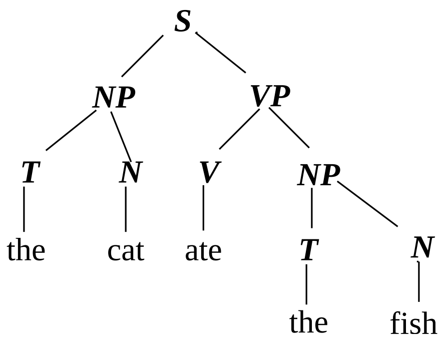
Thus

$$S \overset{*}{\Rightarrow} \text{the} + \text{cat} + \text{ate} + \text{the} + \text{fish}.$$

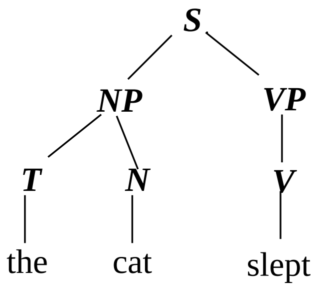
Notation: We use $\Rightarrow$ to mean that one line follows from the previous one by elaborating one of the variables according to a production.

The symbol $\overset{*}{\Rightarrow}$ means the application of a finite sequence of $\Rightarrow$s.

It is much more instructive to draw a **derivation tree**.



Note that we may apply whatever productions we like at each stage:



This grammar can cope with our Nietzschean example. With suitable productions, we have

$$NP \overset{*}{\Rightarrow} \text{the} + \text{man} + \text{who} + \text{said} + \text{that} + S,$$

so

$$\begin{aligned} S &\to NP + VP \\ &\overset{*}{\Rightarrow} \text{the} + \text{man} + \text{who} + \text{said} + \text{that} + S + \text{is} + \text{dead} \\ &\overset{*}{\Rightarrow} \text{the} + \text{man} + \text{who} + \text{said} + \text{that} + \text{the} + \text{man} + \text{who} + \\ &\quad +\text{said} + \text{that} + S + \text{is} + \text{dead} + \text{is} + \text{dead} \\ &\quad \ldots \end{aligned}$$

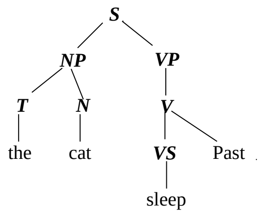
with, at some stage,

$$S \overset{*}{\Rightarrow} \text{God} + \text{is} + \text{dead}.$$

We have evaded the question of tenses. To cope with those, we introduce a non-terminal VS = verb stem and a *terminal* Past, with productions such as

$$V \to VS + \text{Past}, \qquad VS \to \text{sleep, eat, } \textit{et cetera.}$$

The action of the grammar is to produce a sequence



of **morphemes**, 'the cat sleep Past' which, at a later stage, ('morphological rules') is turned into 'the cat slept'.

Transitive and intransitive verbs

Unfortunately our grammar so far allows 'the cat slept the fish'. which is ungrammatical as 'sleep' is an intransitive verb: it cannot have an object. We could avoid this by introducing non-terminal variables V_t (transitive verb) and V_i (intransitive verb). We also introduce an end-of-sentence marker '**.**', which is a terminal. Then we have productions

$$\begin{aligned} S &\to NP + VP + \textbf{.} \\ VP &\to V + NP \\ V + NP &\to V_t + NP \\ V + \textbf{.} &\to V_i + \textbf{.} \\ V_i &\to \text{sleep, walk, } \textit{et cetera} \\ V_t &\to \text{catch, chase, } \textit{et cetera.} \end{aligned}$$

The effect of this (which should be combined with the mechanism for handling tenses) is that a verb followed by a noun phrase must be transitive and a verb at the end of a sentence intransitive.

Consider the productions

$$\begin{aligned} V + NP &\to V_t + NP \\ V + \textbf{.} &\to V_i + \textbf{.} \end{aligned}$$

these are **context-sensitive** productions. They say

$$\begin{aligned} V &\to V_t \\ V &\to V_i \end{aligned}$$

but the first only in the context of being followed by a noun phrase, and the second only in the context of being followed by '**.**'.

By contrast, all the productions we had met previously had been **context-free**: they had a single, non-terminal variable on the left-hand side.

A general **phrase structure grammar**, also called a **type 0 grammar** has productions with almost any terminal and non-terminal variables on either side, the only restriction being that there must be at least one non-terminal somewhere on the left.

A **context-free** or **type 2 grammar** is one all of whose productions are context-free.

A **context-sensitive** or **type 1 grammar** is one all of whose productions are context-sensitive (i.e. context-free productions with a 'context' surrounding either side of the production).

Precise definitions will be given later.

Backus-Naur Form (BNF)

This is a means of defining the syntax of programming languages first devised by John Backus (1959) and Peter Naur (1960) to describe the programming language ALGOL. It is, more or less, a context-free grammar for the language. In fact it contains some extra refinements to make it easier to write the productions.

Notation The symbol `::=` is used in productions where we have used $\rightarrow$.

The symbol $|$ means 'or', which saves writing several very similar productions separately.

Angle brackets $<\ >$ surround the names of non-terminal variables.

Example A program might be described by the production

```
<program> ::=  program
                   <declaration_sequence>
               begin
                   <statements_sequence>
               end ;
```

where `<program>` plays the rôle of S = sentence (the starting variable), and the non-terminals `<declaration_sequence>` and `<statements_sequence>` are elaborated by further productions.

e.g.

```
<statements_sequence> ::=  <statement>|
          <statement>;<statements_sequence>
```

Further extensions are the use of square brackets $[\ldots]$ to indicate optional items, braces $\{\ldots\}$ to indicate items which may be repeated any number of times (i.e. Kleene closure), and various ways of distinguishing terminal variables.

```
<statements_sequence> ::=
      <statement>{";" <statement>}
```

References to documents about BNF and to language specifications using BNF may be found on the course web page.

Phrase Structure Grammars — Formal Definitions

A **phrase structure grammar** is a quintuple

$$G = (V, N, \Sigma, P, S)$$

Where

- $V = N \cup \Sigma$ is a (finite, non-empty) set of **variables**,
- N is the (non-empty) set of **non-terminal variables**,
- $\Sigma \subset V$ is the (non-empty) set of **terminal variables**,
- $S \in N$ is the **starting variable**,
- P is a (finite, non-empty) set of **productions** $\alpha \rightarrow \beta$ with each $\alpha \in V^*NV^*$ and $\beta \in V^*$.

We say $\alpha' \in V^*$ **directly generates** $\beta' \in V^*$, written $\alpha' \Rightarrow \beta'$, if, for some $\alpha_1, \alpha_2 \in V^*$, $\alpha \in V^*NV^*$ and $\beta \in V^*$,

$$\alpha' = \alpha_1\alpha\alpha_2, \qquad \beta' = \alpha_1\beta\alpha_2, \qquad (\alpha \rightarrow \beta) \in P.$$

We say α' **generates** β', written $\alpha' \overset{*}{\Rightarrow} \beta'$, if there is a chain

$$\alpha' = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n = \beta'.$$

The **language defined by** G is

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}.$$

Two grammars G_1, G_2 are **equivalent** if $L(G_1) = L(G_2)$.

Example Consider the grammar G_1 with

1. $N = \{S, A, B\}$
2. $\Sigma = \{a, b\}$
3. starting variable S
4. productions

$$S \rightarrow aA, \quad A \rightarrow aAB, \quad B \rightarrow b, \quad A \rightarrow a$$

Then we get derivations such as:

$$S \to aA \Rightarrow aaAB \Rightarrow aaaABB \Rightarrow aaaaBB \Rightarrow$$
$$\Rightarrow aaaaBb \Rightarrow aaaabb,$$

so that $S \overset{*}{\Rightarrow} aaaabb$.

Productions

$$S \to aA, \qquad A \to aAB, \qquad B \to b, \qquad A \to a.$$

Generally, a derivation has to start with $S \to aA$, and the A will have to produce an a eventually, so all strings in $L(G)$ contain at least two as. Production of any further as depends on the use of $A \to aAB$, which ultimately leads to an equal number of bs at the end of the string.

$$L(G_1) = \{a^{n+2}b^n : n \geq 0\}.$$

Example Consider the grammar G_2 with

1. $N = \{S, A\}$
2. $\Sigma = \{a, b\}$
3. starting variable S
4. productions

$$S \to aaA, \qquad A \to aAb, \qquad A \to \varepsilon$$

Clearly this operates in a similar way to G_1, but producing bs directly, rather than through Bs, and producing two as in the first move, but allowing A to just vanish.

$$L(G_2) = \{a^{n+2}b^n : n \geq 0\}.$$

The grammars G_1 and G_2 are equivalent. We are interested in grammars with various restrictions on the productions. In this context, a general phrase structure grammar as defined above will also be called a **Type 0 grammar**.

A **context-free grammar**, or **Type 2 grammar**, is one in which all the productions are of the form

$$A \to \beta, \qquad \text{with } A \in N,\ \beta \in V^*.$$

Example Consider the language L over the alphabet $\Sigma = \{\text{‘(’}, \text{‘)’}\}$ consisting of just those *non-empty* strings of brackets in which the number of opening brackets equals the number of closing brackets and such that in no initial segment of the string does the number of closing brackets exceed the number of opening brackets. Find a context-free grammar for L.

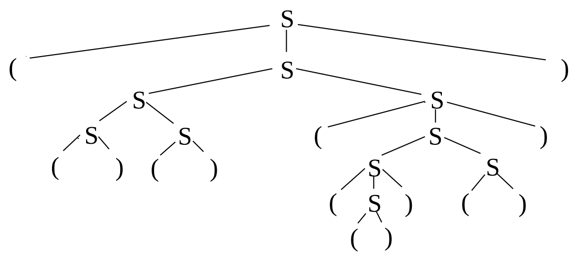
Let S be our starting symbol and the sole non-terminal variable and let the productions be

$$\begin{aligned} S &\to (\,) \\ S &\to (S) \\ S &\to SS \end{aligned}$$

For example, we obtain the string ‘(( ) ( ) ((( )) ( )))’ by

$$\begin{aligned} S &\to (S) \\ &\Rightarrow (SS) \\ &\Rightarrow (SSS) \\ &\overset{*}{\Rightarrow} ((\,)(\,)(S)) \\ &\Rightarrow ((\,)(\,)(SS)) \\ &\overset{*}{\Rightarrow} ((\,)(\,)((S)(\,))) \\ &\Rightarrow ((\,)(\,)(((\,))(\,))) \end{aligned}$$

The derivation tree looks like this.



If we wanted L to be the language of *possibly empty* strings of brackets satisfying the same conditions, we could achieve this with the productions

$$\begin{aligned} S &\to \varepsilon \\ S &\to (S) \\ S &\to SS \end{aligned}$$

where we can retrieve $S \to (\,)$ because

$$S \to (S) \Rightarrow (\varepsilon) = (\,)$$

Example Find a context-free grammar G with $\Sigma = \{a, b, c\}$ such that

$$L(G) = \{a^n b^m c^n : n \geq 3,\ m \geq 0\}.$$

Let us start with $N \supseteq \{S, S_1, S_2, S_3\}$, start symbol S and productions

$$S \to aS_1c, \qquad S_1 \to aS_2c, \qquad S_2 \to aS_3c.$$

There is only one possible derivation:

$$S \to aS_1c \;\Rightarrow\; a^2S_2c^2 \;\Rightarrow\; a^3S_3c^3$$

We now need a grammar with start symbol S_3 which generates the language

$$\{a^\ell b^m c^\ell : \ell \geq 0,\ m \geq 0\}.$$

We can get the as and cs in equal numbers by adding the production

$$S_3 \to aS_3c$$

We then switch to producing bs by introducing another non-terminal variable B say with productions

$$S_3 \to B, \qquad B \to bB, \qquad B \to \varepsilon$$

This completes the desired grammar: $N = \{S, S_1, S_2, S_3, B\}$, the start symbol is S and the full list of productions is:

$$S \to aS_1c, \qquad S_1 \to aS_2c, \qquad S_2 \to aS_3c,$$

$$S_3 \to aS_3c \qquad S_3 \to B, \qquad B \to bB, \qquad B \to \varepsilon.$$

Between types 0 and 2 comes the following.

A **context-sensitive grammar**, or **Type 1 grammar**, is one in which all the productions are of the form either

$$\alpha A\gamma \to \alpha\beta\gamma, \qquad \text{with } A \in N,\ \alpha, \gamma \in V^*,\ \beta \in V^+.$$

(i.e. $A \to \beta$ in the context $\alpha \ldots \gamma$)
or $S \to \varepsilon$, though if this occurs then S must not occur on the right-hand side of any production.

[You do not need to know about context-sensitive grammars for this course: they are included here to help explain the term 'context-free'.]

A **right-linear grammar** is one in which all the productions are of the form

$$A \to w, \qquad A \to wB, \qquad \text{with } A, B \in N,\ w \in \Sigma^*.$$

A **left-linear grammar** is one in which all the productions are of the form

$$A \to w, \qquad A \to Bw, \qquad \text{with } A, B \in N,\ w \in \Sigma^*.$$

A **regular grammar**, or **Type 3 grammar**, is one which is either right-linear or left-linear.

Theorem

The set of languages generated by right-linear grammars
is equal to
the set of languages generated by left-linear grammars
and these languages are precisely the regular languages;
i.e. the languages defined by regular expressions,
i.e. the languages accepted by finite automata.

Example Consider the right-linear grammar G_3 with $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, starting variable S and productions

$$S \to aA, \quad A \to aA, \quad A \to aB, \quad B \to abB,$$

$$B \to b, \quad A \to a$$

Then we get derivations such as:

$$S \to aA \;\Rightarrow\; aaA \;\Rightarrow\; aaaA \;\Rightarrow\; aaaa$$

$$S \to aA \;\Rightarrow\; aaA \;\Rightarrow\; aaaB \;\Rightarrow\; aaaabB \;\Rightarrow$$

$$\Rightarrow\; aaaababB \;\Rightarrow\; aaaababb$$

Generally, one can see that

$$L(G_3) = \{a^n : n \geq 2\} \cup \{a^n(ab)^m b : n \geq 2,\ m \geq 0\};$$

i.e. it corresponds to the regular expression

$$\boldsymbol{aa}^*(\boldsymbol{a} \cup \boldsymbol{a}(\boldsymbol{ab})^*\boldsymbol{b})$$

Examples In the earlier examples above, G_1, with productions

$$S \to aA, \qquad A \to aAB, \qquad B \to b, \qquad A \to a$$

is context-free, but not regular, since the production $A \to aAB$ is neither right- nor left-linear.

The (equivalent) grammar G_2 with productions

$$S \to aA, \qquad A \to aAb, \qquad A \to a$$

is also context-free, but not regular, since the production $A \to aAb$ is neither right- nor left-linear.

Example Consider the grammar G_4 with $N = \{S, A\}$, $\Sigma = \{a, b, c\}$, starting variable S and productions

$$S \to aA, \quad A \to Sb, \quad S \to c$$

Notice that G_4 is neither left-linear nor right-linear, so it is not regular.

All derivations are of the form:

$$S \to aA \;\Rightarrow\; aSb \;\Rightarrow\; aaAb \;\Rightarrow\; aaSbb \;\Rightarrow\; \ldots$$

$$\ldots \;\Rightarrow\; a^nSb^n \;\Rightarrow\; a^ncb^n$$

$$L(G_4) = \{a^ncb^n : n \geq 0\}.$$

We recognize this as a language which is not regular.

Example Consider the grammar G_5 with $N = \{S, A\}$, $\Sigma = \{a, b, c\}$, starting variable S and productions

$$S \to aA, \quad S \to A, \quad A \to Sb, \quad A \to S, \quad S \to c$$

Again, G_5 is neither left-linear nor right-linear, so it is not regular.

All derivations are similar to those of G_4 except that the *a*s and *b*s are not necessarily produced equally as the non-terminal variable switches between A and S: e.g.

$$S \to aA \;\Rightarrow\; aS \;\Rightarrow\; aaA \;\Rightarrow\; aaSb \;\Rightarrow\; aacb$$

Generally

$$L(G_5) = \{a^ncb^m : n, m \geq 0\}.$$

$$L(G_5) = \{a^ncb^m : n, m \geq 0\}.$$

We recognize this as a regular *language*, corresponding to the regular expression

$$\boldsymbol{a^*cb^*}.$$

A **regular language** is one which can be produced by a regular grammar. This example shows that it may also be produced by a non-regular grammar.

Problem. Find a regular grammar G_6 with $L(G_6) = L(G_5)$.

Answer. Let G_6 be the grammar with $N = \{S, B\}$, $\Sigma = \{a, b, c\}$, starting variable S and productions

$$S \to aS, \quad S \to cB, \quad B \to bB, \quad B \to \varepsilon$$

Pushdown Automata

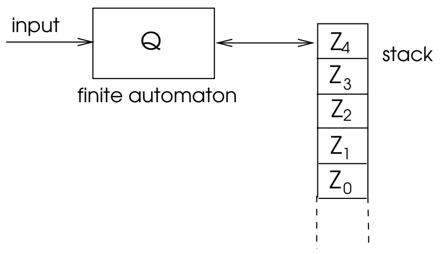
Regular languages correspond to the concept of computation embodied in the idea of a finite automaton.

To what do context-free grammars correspond?

The answer is the **non-deterministic pushdown automata** (NPDA).

Basically, these are finite automata with an infinite memory, **but** with the memory organized in the form of a pushdown stack; i.e. a last-in-first-out (LIFO) store.

A pushdown automaton



Definition An NPDA is a septuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

1. Q is a finite **set of states**;
2. Σ is the **input alphabet**;
3. Γ is the **stack alphabet**;
4. $q_0 \in Q$ is the **initial state**;
5. $Z_0 \in \Gamma$ is the **start symbol**, indicating the bottom of the stack;
6. $F \subseteq Q$ is the **set of final states**;
7. δ is a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

To understand this, consider the special case of a **deterministic pushdown automaton**. In this,

$$\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*.$$

When the machine is in state q and receives an input a, it looks at the character on the top of the stack. If this character is Z, then δ specifies its course of action. Suppose

$$\delta(q, a, Z) = (p, \gamma).$$

Then the machine moves to state p and records the string γ in place of Z at the top of the stack (the rightmost symbol of γ being placed lowest).

The machine accepts an input if at the end of the input the machine is in one of the final states. (An alternative, and equivalent, model accepts if the stack is empty.)

Non-deterministic PDAs have

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}.$$

They can make any one of the moves defines by the (p_i, γ_i). Thus the progress of the machine is not completely determined: from a given situation, with given inputs, the machine has a variety of possible futures. A string is accepted if it is accepted in any one of these possible futures.

A further refinement is that the machine can make moves without receiving any input (**ε-moves**). If

$$\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\},$$

then the machine can make any of the moves specified by the (p_i, γ_i).

Theorem The set of strings accepted by a NPDA is a context-free language and every context-free language is the set of strings accepted by some NPDA.

Remarks

- From a practical, compiler designer's, point of view, it is more convenient to work with languages accepted by deterministic PDAs. These are called the **deterministic context-free languages** and it is possible to define classes of grammars which define (some) such languages. The deterministic context-free languages do form a proper subset of the context-free languages.
- There is a similar notion of **non-deterministic finite automata** (NFA), but they accept the same class of languages as do the deterministic finite automata. However, NFAs are needed to prove Kleene's Theorem.

- In another direction, characterizations of type 0 and type 1 languages may be given using more general models of computation (Turing machines).
- A level 2 course 'Machines and Languages' studies these matters further.

END OF LECTURE COURSE

MERRY

CHRISTMAS!