Slide 1

Foundations of Computer Science

Contents

Slide 2

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

Slide 3

Conclusion

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

But

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

Slide 4

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 its function correctly.

Slide 5

1. Formal specification

- Natural language (English, French, etc.) is not logical :

  Sheffield does not have anything that Manchester does not have.

  but mathematics is logical.

- Natural language has flaws, such as contradictions :

  Put all the shopping in one bag but don't make it too heavy.

  but mathematical logic reveals these flaws.

Slide 6

2. Mathematical verification

- A computer program cannot be tested exhaustively
- We can write a formal specification in logic of what the progran *should* do, and test the specification mathematically by the verification of necessary properties.

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.$\square$

Slide 7

Summary :

- Many systems are necessarily complex, and must interact with ill–understood environments.

Requirements :

- 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.

Slide 8

Set Theory

Set theory is a *universal language* that is very close to *natural language* with a very simple notation. It provides an infrastructure that allows both description and reasoning about systems and their behaviour.

It is

- simple
- concise
- unambiguous
- accurate

Slide 9

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 data bases involve set theory.

The methods of proof that have been considered dealt with numbers. However in practice it is necessary to prove statements about other 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.

Slide 10

Definitions :

1. A *set* is a well-defined list or 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
$$B = \{x : x \text{ is an integer, } x > 0\}.$$

Slide 11

Properties of sets :

1. Order is not important. Thus
$$\{a, 2, red\} \text{ and } \{2, a, red\}$$
are the same set.
2. The frequency of occurrence of the elements of a set is not important. Thus
$$\{a, 2, red\} \text{ and } \{2, a, red, 2, 2, a, 2\}$$
represent 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.

Slide 12

Some standard sets

Some commonly used sets are

1. $B = \{true, false\}$ is the set of *Boolean values.*
2. $N = \{0, 1, 2, 3, \ldots\}$ is the set of *natural numbers.*
3. $Z = \{0, \pm1, \pm2, \pm3, \ldots\}$ is the set of *integers.*
4. R is the set of real numbers.

The sets N, Z and R are infinite.

Slide 13

Subsets

If every element of set A is also an element of another set B, then A is a *proper subset* of B, $A \subset B$, or A is a *subset* of B, $A \subseteq B$.

Example If

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

then $A \subset B$. Equality of the sets can never be achieved because the set B has $green$ as one of its elements. □

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. Thus $A = B$. □

Slide 14

Equality of sets

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 $B \not\subseteq A$. □

Slide 15

When is $A \subseteq B$ satisfied ?

Example Let A be the set of all people who live in a region, 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 $A \subset B$

Thus $A \subseteq B$, depending on extra conditions. □

Slide 16

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 strings or sets of characters

Slide 17

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 existence of $\emptyset$ cannot be proved; it is an axiom of set theory. It is accepted either because it is obvious, or because it reflects a certain view that is not contradicted by our intuition.
- The empty set is unique
- The empty set $\emptyset$ is a subset (not element) of every set A.

Slide 18

Examples

1. $\emptyset = \{x | x \neq x\}$
2. $\emptyset = \{n | n \in \mathbb{N} \wedge n > n\}$
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

Slide 19

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*, also called a *class of sets* or a *collection of sets*.

Example Let $A = \{1, 2, 3, 4\}$ and let B be the class 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 class of subsets of B that contain 2 and two other elements of A

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

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

Slide 20

For every set A, we maybe interested in the class of all subsets of A. This is called the *powerset 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, called its *cardinality*, then the number of elements in $P(A)$ is 2^n.

Slide 21

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.

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 :

$$A^c = \{x : x \in U \text{ and } x \notin A\}.$$

Slide 22

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

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

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

Slide 23

Note the distinction :

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

Example

The following statements are true :

$N \subseteq Z, \quad N \subset Z, \quad \{1, 3, 7\} \subseteq N, \quad \{1\} \subseteq R, \quad 1 \in R,$

$\emptyset \subset N, \quad \emptyset \subset \{0\}, \quad -5 \in Z$

The following statements are false :

$1 \subset R, \quad 0 \subset \emptyset, \quad \{0\} \subset \emptyset, \quad \{3, 4\} \in N, \quad \{5\} \notin N$

$\square$

Slide 24

Venn diagrams

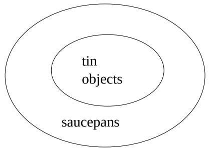
- A Venn diagram is a pictorial representation of sets by subsets of points in the plane.
- The universal set U is represented by the interior of a rectangle, and the other sets are represented by disks lying within the rectangle.
- Many verbal statements can be translated into equivalent statements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

Slide 25

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

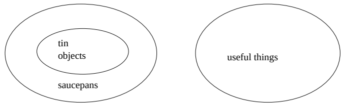
1. S_1 : My saucepans are the only things that I have that are made of tin.
2. S_2 : All your presents are useful.
3. S_3 : None of my saucepans is of the slightest use.
4. C : The presents that you give me are not made of tin.

The Venn diagram for S_1 is

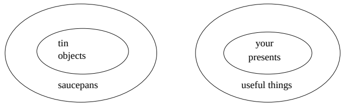


Slide 26

The Venn diagram for S_3 is

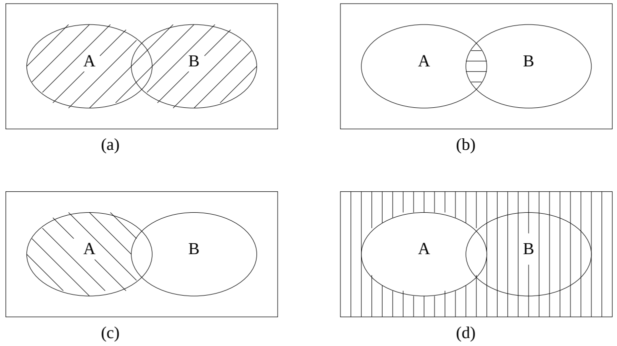


The Venn diagram for S_2 is



Thus the conclusion is true.$\square$

Slide 27



(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.

Slide 28

The laws of set algebra

1. Idempotent Laws :

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

2. Associative Laws :

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

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 :

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

Slide 29

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 :

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

Slide 30

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 $x \notin B$, or $x \in A^c$ 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 $x \notin B$, or $x \notin A \cup B$. Thus $x \in (A \cup B)^c$.

The result follows from the combination of the two results. $\square$

Slide 31

Use the laws of set algebra to simplify set expressions.

Example

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

Slide 32

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 $(A \cap B) \subseteq A \subseteq (A \cup B)$.

Satisfaction of equality :

1. If $B = \emptyset$, then $\emptyset \subseteq A = A$.
2. If $B = U$, then $A = A \subseteq U$. $\square$

Slide 33

Functions

A *function* extends the notion of a relation between sets.

Example Consider a database for recording (1) vehicles and their registration numbers and (2) vehicle registration numbers and vehicle owners.

This *abstraction* of a system for vehicle ownership and registration is not complete because every vehicle can have only one registered owner, and a registration number is a unique identifier of a vehicle. A *function* extends the notion of relationship between sets by restricting the association of one element in a set to no more than one element in another set. □

Slide 34

Definition :

A *function* f is 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

A telephone directory is a function that ascribes to each person exactly one telephone number. The domain of the function is a list of people and the co–domain is the set of telephone numbers. □

Example

Since nobody can have more than one birthday, a mapping between a group and their birthdays defines a function. □

Slide 35

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 into the element y of B.
3. $f(x)$ denotes the element of B that is the mapping of the element $x \in A$ by the function f.

Example

Let f be the function from R to R that maps every real number to its square. This function may be written as

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

□

Slide 36

Example

Let $A = \{1, 2, 3, 4\}$ and $B = \{a, b, c, d, e\}$. If the function f is defined by

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

then the domain of f is A and the co-domain of f is B.□

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 mappings 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 mapping of more than one element of the domain.

Slide 37

- 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 :

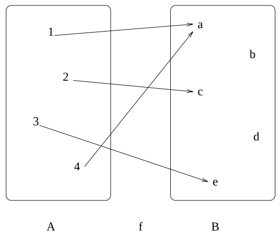
$$range(f) = \{f(a) : a \in A\}$$

- If $f : A \to B$ and $a \in A$, then the unique element of B which the function f assigns to a is called the *image* of a under f, or the value of f at a, and is denoted by $f(a)$.

Slide 38

Example

For the sets in the previous example



$domain(f) = \{1, 2, 3, 4\}$

$codomain(f) = \{a, b, c, d, e\}$

$range(f) = \{a, c, e\}$ $\square$

Slide 39

One–to–one, onto and invertible functions

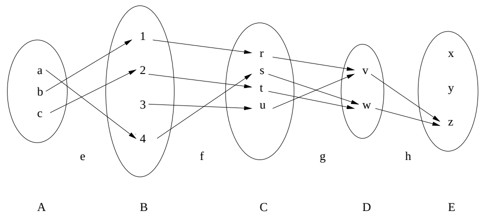
Definitions :

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.

Slide 40

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. $\square$

Slide 41

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.

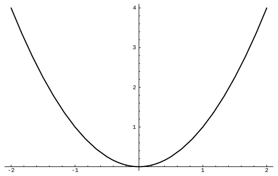
Slide 42

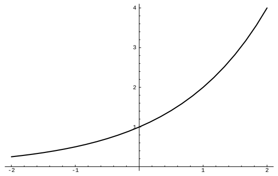
Example

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.□

Slide 43

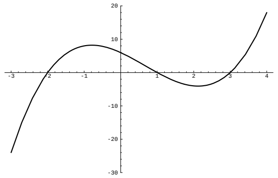


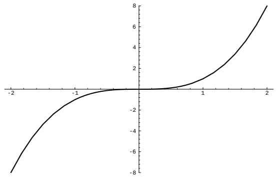


$e : R \to R; e(x) = x^2$ is neither 1–1 nor onto.

$f : R \to R; f(x) = 2^x$ is 1–1 and not onto.

Slide 44





$g(x) = x^3 - 2x^2 - 5x + 6$ is onto but not 1–1.

$h(x) = x^3$ is 1–1 and onto.

Slide 45

Example

Show that the function $f : R \to R$, given by $f(x) = x^3 + 1$, 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

$$x_1^3 + 1 = x_2^3 + 1$$

implies that $x_1^3 = x_2^3$ which implies that $x_1 = x_2$ since every real number has a unique real cube root :

$$x^3 - y^3 = (x - y)(x^2 + xy + y^2)$$

and the roots of $(x^2 + xy + y^2) = 0$

are imaginary

Slide 46

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

$$x^3 + 1 = y$$

has a solution $x \in R$ for every $y \in R$. Since

$$x = (y - 1)^{\frac{1}{3}}$$

it follows that f is onto : For every value of y, there is a unique solution x.

Since f is 1–1 and onto, it is also invertible.

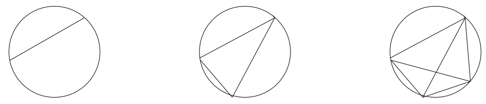
□

Slide 47

Mathematical thinking – a logical approach for enquiry

Example

Consider a circle with n points on its circumference :



2 points 3 points 4 points

2 regions 4 regions 8 regions

Slide 48

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

Conclusion : The maximum number of possible regions = 2^{n-1} where n is the number of points.

But : This formula fails for $n = 6$. □

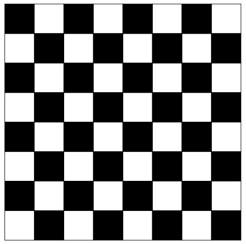
Slide 49

Example

A chess board has 64 squares.

How many 2×2 squares are there ?

How many 3×3 squares are there ?



Slide 50

size of square	number of squares
1×1	$64 = 8^2$
2×2	$49 = 7^2$
3×3	$36 = 6^2$
$\dots$	$\dots$
8×8	$1 = 1^2$

Slide 51

Conclusion : The total number of squares of size $n \times n$ is $(9-n)^2$

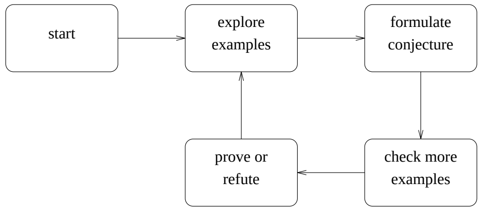
Problem : What happens if $n = 10, \quad 50, \quad 100$?

Conjecture : The total number of squares of size $n \times n$ is

$$1^2 + 2^2 + \cdots + (n-1)^2 + n^2. \qquad \square$$

Slide 52

This is the typical investigation cycle :



but nothing has been formally proven

Slide 53

Proof

Types of Proof :

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

Slide 54

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 $a = b$.

Slide 55

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 $a = b$. $\square$

Slide 56

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

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

Slide 57

Since

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

and this formula must be checked for $r = 0, 1, 2, 3, 4$.

The first term on the right hand side is divisible by 5, and the second term is not divisible by 5. $\square$

Slide 58

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.

$\square$

Slide 59

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

$$ax_1 + b = c$$
$$ax_2 + b = c$$

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. $\square$

Slide 60

4. Proof by induction

This method of proof is useful for the establishment of the truth of propositions that involve an infinite number of 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. $\square$

Slide 61

The principle of mathematical induction :

If we can prove that **the result is true for a specific value of n, say $n = k$**, and **if we assume that it is true for an arbitrary value of $n \geq k$**, then we can prove that **it is true for the next value of n**.
We then deduce that the result is true for all values of $n \geq k$.

Three points :

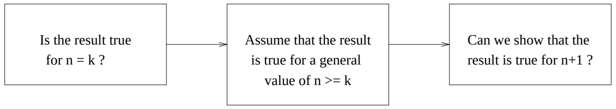
1. The specific value of n is *usually* 1 or 2.
2. It is necessary that $n \geq k$; $n > k$ is not adequate.
3. Both parts of the definition are important;

   we need both a starting point (the first part or base case), and the second part, called the inductive step, to be true.

Slide 62

The steps involved in induction are :

1. Find a value of k for which the result works.
2. Assume that the result is valid for an arbitrarily chosen value of $n \geq k$.
3. Prove that the result also holds for the case $n + 1$.
4. Conclude that the result is valid for all $n \geq k$.



Slide 63

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

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

Step 1. We must show that there exists an integer for which the theorem is true. The simplest example is $n = 1$. Clearly, the sum of the first positive integer is 1, and

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

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

Slide 64

Step 2. We now assume that the theorem is true for a given arbitrary value of $n = m$, and must show that it is true for $n + 1 = m + 1$.

Sum of first $m + 1$ positive integers = sum of first m positive integers + $(m + 1)$.

By assumption,

$$\text{Sum of first } m \text{ positive integers} = \frac{m(m+1)}{2}$$

and thus

$$\text{Sum of first } m + 1 \text{ positive integers} = \frac{m(m+1)}{2} + (m + 1) = \frac{(m+1)(m+2)}{2}$$

which is the general situation with n replaced with $m + 1$.

We have shown

1. The statement is true for $n = 1$.
2. If the statement is true for $n = m \geq 1$, then the statement is true for $n = m + 1$. $\square$

Slide 65

Example

Theorem : $5^n - 4n - 1$ is divisible by 16 for all integers $n \geq 2$. (It is trivially true for $n = 1$.)

Proof : Define

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

Then

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

□

Slide 66

Example

Theorem : $n^3 - 4n - 6$ is divisible by 3 for all $n \in Z$.

Proof : Define

$$S(n+1) = (n+1)^3 - 4(n+1) - 6$$

Then for $n \in Z$

$$\begin{aligned} S(n+1) &= (n+1)^3 - 4(n+1) - 6 \\ &= n^3 + 3n^2 - n - 9 \\ &= n^3 - 4n - 6 + 3n^2 + 3n - 3 \\ &= S(n) + 3(n^2 + n - 1) \end{aligned}$$

□

Slide 67

Example

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

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

where $a, b \geq 0$.

Proof : We first note the requirement that $n \geq 14$.

The only numbers less than 14 that can be written in the form $3a+8b$ are $3, 6, 8, 9, 11$ and 12. Thus only some of the numbers less than 14 can be written in the required form. It must be shown that all numbers greater than or equal to 14 can be written in this form.

Step 1. We must show that there exist constants a and b such that $3a + 8b = 14$. Clearly, this equation is satisfied by $a = 2, b = 1$.

Slide 68

Step 2. We assume that the theorem is true for an integer $m \geq 14$, that is, there exist constants a_1 and b_1 such that

$$3a_1 + 8b_1 = m,$$

and must show that the equation

$$3a_2 + 8b_2 = m + 1,$$

also has a non-negative solution. Since

$$m + 1 = 3a_1 + 8b_1 + 1 = 3\left(a_1 + 3\right) + 8\left(b_1 - 1\right),$$

we have

$$a_2 = a_1 + 3 \text{ and } b_2 = b_1 - 1,$$

and the conditions of the theorem are satisfied if $b_1 \neq 0$.

Slide 69

If $b_1 = 0$, then $3a_1 = m > 14$, and thus there exists an integer $c \geq 0$ such that

$$a_1 = 5 + c.$$

Hence

$$m + 1 = 3a_1 + 1 = 3(5 + c) + 1 = 3c + 8.2$$

which is of the required form.

We have shown

1. The statement is true for $n = 14$.
2. If the statement is true for $n = m \geq 14$, then the statement is true for $n = m + 1$. $\square$

Slide 70

Recursion

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. The problem 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. $\square$

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

Slide 71

Definition :

A *recursive function* f is a function whose domain includes f.

- Many programming languages are recursive because they allow procedures to call themselves.
- Recursively defined sequences occur frequently in practice.

A sequence of numbers is defined recursively if

- Requirement (Base values)

  a finite set of values is specified.
- Requirement (Remaining values)

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

Slide 72

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. 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. $n = 4 : FACT(4) = 4 \times FACT(3) = 24.$

Slide 73

The following procedure implements the factorial function recursively.

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}
```

□

Slide 74

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

Slide 75

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
```

□

Slide 76

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.

Slide 77

Example Writing a string of characters backwards

- The base case is either the empty string or the string of length 1.
- What is the simpler problem that generates the recursion ?

  There are two options :

  – Remove the last character of the string, or

  – Remove the first character of the string.

Let the string be S of length size where size ≥ 0.

Slide 78

If the last character of the string is removed :

procedure WriteBackward (S : string; size : integer);

begin

  if size > 0

    then begin

      Write (S[size]); $\{$*write the last character*$\}$

      WriteBackward (S, size-1) $\{$*write the rest of the string backward*$\}$

    end

    $\{$*size = 0, the base case*$\}$

end $\{WriteBackward\}$

Slide 79

If the first character of the string is removed :

procedure WriteBackward (S : string; size : integer);

begin

  if size > 0

    then begin

      WriteBackward (S, size-1) $\{$*write the rest of the string backward*$\}$

      Write (S[size]); $\{$*write the first character*$\}$

    end

    $\{$*size = 0, the base case*$\}$

end $\{WriteBackward\}$

□

Slide 80

Example

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

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

The number of digits in 470 is

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

□

Slide 81

Induction and recursion

Induction and recursion are closely related :

1. Base values
   - In induction, the proposition is assumed known for a specific value of n.
   - In recursion, a finite number of values of the sequence are assumed to be known.
2. Remaining values
   - In induction, we assume that the proposition is true for the value n, and prove that it is true for $n+1$.
   - In recursion, the nth element of the sequence is used to calculate the $(n+1)$th element of the sequence.

Induction is required to prove properties about *recursive* programs.

Slide 82

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
$$S = \{S_1, S_2, S_3, S_4, \dots\}.$$
- Recursion starts from the sequence S where
$$S_n = S_{n-1} + n.$$
The recursion stops at the base value $S_1 = 1$. This is used to construct an expression for the function f that assigns to each number n, the value S_n. $\square$

Slide 83

Propositional Logic

Propositional calculus is required for

- model reasoning – its original purpose
- reasoning by computer, eg, theorem provers, program verifiers
- digital circuit design
- data model for programming languages and systems, eg, Prolog
- the recognition of valid and invalid arguments

Propositional calculus is the study of logical relationships between objects called propositions.

Slide 84

What is propositional logic ?

Consider the statement

if $(a < b \;\; || \;\; (a >= b \;\; \&\& \;\; c == d))\dots$

Note :

1. $||$ denotes OR. Evaluate the second expression only if the first expression is FALSE
2. $\&\&$ denotes AND. Evaluate the second expression only if the first expression is TRUE

The expression simplifies to

if $(a < b \;\; || \;\; c == d))\dots$

Slide 85

Properties of propositions :

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

Slide 86

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), (5) and (6) are not propositions.

□

Slide 87

What are the limitations of propositional logic ?

Consider the expression

`if` $(a < b \quad \&\& \quad a < c \quad \&\& \quad b < c) \ldots$

This is equivalent to

`if` $(a < b \quad \&\& \quad b < c) \ldots$

but it cannot be deduced by propositional logic because it contains *predicates*, that is, expressions that contain arguments that may be true or false, depending on the values assigned to the arguments

Slide 88

Composite statements and logical connectives

Definitions

- Compund 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. □

Slide 89

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 can be defined by enumerating all the possibilities, and this can be done by a *truth table*.

Slide 90

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

Table 1: The truth table for the conjunction connective.

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

Slide 91

2. Disjunction : Two statements can be combined with the word *or* to form a compound statement of the original statements.

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

Table 2: The truth table for the disjunction connective.

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

Slide 92

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

Table 3: The truth table for the conditional implication connective.

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

Slide 93

The last two propositions suggest that false propositions can imply anything because the conditional implication is defined to be true if p is false.

Consider the statements :

- *If it will rain on Tuesday, John will take his umbrella.*

  If it does not rain on Tuesday, John may or may not take his umbrella; the statement is true but not informative.

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

  If the power indicator light is lit, then the unit must be connected to the supply, and so $p \to q$ is true.

  If the power indicator light is off, then the unit may be connected to the supply, but another fault may be apparent.

Slide 94

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

p	$\neg p$
T	F
F	T

Table 4: The truth table for the negation connective.

Slide 95

5. Equivalence : The equivalence connective, also called the biconditional connective, $p \Leftrightarrow q$ is defined by the truth table for $p \to q \wedge q \to p$.

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

Table 5: The truth table for the biconditional connective.

Slide 96

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

Table 6: The truth table for the biconditional connective.

This proposition is expressed linguistically by preceding either component by *if and only if*, and the other proposition may then be preceded by *then.*

Example

(a) If and only if you clean the car, then I'll give you $£10$.

(b) I'll give you $£10$ if and only if you clean the car. $\square$

Slide 97

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

Example

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

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

□

Slide 98

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 □

Slide 99

Example Suppose that S, W, R and U are true.

(a) Show that $(S \to W) \wedge (\neg R \wedge U)$ is false.

This reduces to $(T \to T) \wedge (F \wedge T) = T \wedge F = F$.

(b) Show that $(\neg(R \wedge U)) \to ((\neg W) \wedge S)$ is true.

This reduces to $(\neg(T \wedge T)) \to (F \wedge T) = F \to F = T$.

□

Slide 100

Rules for simplification

We can use the following rules to simplify expressions that involve logical connectives.

- Commutative laws :

$$\begin{aligned} P \vee Q &= Q \vee P \\ P \wedge Q &= Q \wedge P \end{aligned}$$

- Associative laws :

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

Slide 101

- Distributive laws :

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

- De Morgan's laws :

$$\begin{aligned} \neg(P \vee Q) &= \neg P \wedge \neg Q \\ \neg(P \wedge Q) &= \neg P \vee \neg Q \end{aligned}$$

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

Slide 102

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

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

Slide 103

A *contradiction* is 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. $\square$

Slide 104

The Principles of 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.
- If a compound proposition P contains a proposition Q, and if Q is replaced by a logically equivalent proposition Q^*, then the resulting proposition P^* is logically equivalent to P.

These rules enable logical equivalences and implications to be obtained without using truth tables.

Slide 105

Example

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

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

$$[(q - r) \wedge ((q - r) \to q)] \to q$$

is obtained.

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

$$[p \wedge (p \to (q - r))] \to (q - r)$$

is obtained. $\square$

Slide 106

Example

Construct a proposition that is logically equivalent to $(p \wedge q) \to (\neg p \wedge q)$ that does not use the connective $\wedge$.

Since $p \wedge q$ is logically equivalent to $\neg(\neg p \vee \neg q)$, and $\neg p \wedge q$ is logically equivalent to $\neg(\neg\neg p \vee \neg q)$, the given proposition is equivalent to

$$\neg(\neg p \vee \neg q) \to \neg(\neg\neg p \vee \neg q)$$

and thus to

$$\neg(\neg p \vee \neg q) \to \neg(p \vee \neg q)$$

which is equivalent to

$$\neg(p \to \neg q) \to \neg(q \to p)$$

$\square$

Slide 107

Tautologies and a Valid Proof

A *theorem* consists of a series of propositions $H_1, H_2, \ldots H_n$, called *premises*, and a proposition C, called a *conclusion*.

A theorem is *valid* if C is true whenever all the premises $H_1, H_2, \ldots, H_n$ are true

$$H_1 \wedge H_2 \wedge \cdots \wedge H_n \Rightarrow C$$

$P \Rightarrow Q$ denotes P **logically implies** Q, that is,

- If P is true, then Q
- If P is false, then Q is either true or false
- P is true and Q is false is not allowed

Slide 108

Consider the implication connective for the three allowable cases

P	Q	$P \to Q$
T	T	T
F	T	T
F	F	T

that is, $P \to Q$ *is a tautology*.

Thus the theorem

$$H_1 \wedge H_2 \wedge \cdots \wedge H_n \Rightarrow C$$

is valid *if and only if*

$$H_1 \wedge H_2 \wedge \cdots \wedge H_n \to C$$

is a tautology.

Slide 109

Example

Show that $p \Leftrightarrow q$ logically implies $p \rightarrow q$.

p	q	$p \Leftrightarrow q$	$p \rightarrow q$	$(p \Leftrightarrow q) \rightarrow (p \rightarrow q)$
T	T	T	T	T
T	F	F	F	T
F	T	F	T	T
F	F	T	T	T

The truth table shows that if $p \Leftrightarrow q$ is true, then $p \rightarrow q$ is true, but if $p \Leftrightarrow q$ is false, then $p \rightarrow q$ may be true or false. $\square$

Slide 110

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 is based on inference rules.

Slide 111

Inference rules

Equivalence :

If two propositions are equivalent, then one of the propositions can be replaced by the other proposition at any point in a proof.

Implication :

Example

If $p \wedge q$ is true, we can deduce q because $(p \wedge q) \rightarrow q$ is a tautology.

Equivalences and implications can be combined to prove a conclusion from hypotheses.

Slide 112

Example

Suppose the equivalences

1. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
2. $p \wedge p \equiv p$

are known, and it is required to prove that $a \wedge (a \vee b) \equiv a \vee (a \wedge b)$.

1. $a \wedge (a \vee b) \equiv (a \wedge a) \vee (a \wedge b)$ from (1).
2. $(a \wedge a) \vee (a \wedge b) \equiv a \vee (a \wedge b)$ from (2). $\square$

Slide 113

Example

Suppose it is known that $p \wedge (q \wedge r)$ is true, and it is desired to know if r is true.

1. The hypothesis H is $p \wedge (q \wedge r)$ and is assumed to be true.
2. Since $a \wedge b \rightarrow b$ is a tautology, we deduce that

   $p \wedge (q \wedge r) \rightarrow q \wedge r$

   by replacing b by $q \wedge r$.
3. Repeat this step to deduce that r is true.

   $q \wedge r \rightarrow r$
4. Conclusion

   $p \wedge (q \wedge r) \rightarrow q \wedge r \rightarrow r$

   $\square$

Slide 114

Inference rules

An inference rule is a list of propositions above a horizontal line and one proposition below the line. It may be used to

1. eliminate a logical connective, or
2. introduce a logical connective

The proposition above the line is assumed to be true, and the proposition below the line can be deduced from it.

Thus

if a proof involves an expression that contains the formula above the line, then it can be replaced by the formula below the line

Slide 115

Example

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

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.$\square$

Example

Elimination rules for conjunction

$$\frac{p \wedge q}{p} \;\; [\wedge E1], \qquad \frac{p \wedge q}{q} \;\; [\wedge E2].$$

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. $\square$

Slide 116

Example

Introduction rule for conjunction

$$\frac{p \quad q}{p \wedge q} \;\; [\wedge I].$$

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

I denotes that this is an introduction rule. $\square$

Example

Introduction rules for disjunction

$$\frac{p}{p \vee q} \;\; [\wedge I1], \qquad \frac{q}{p \vee q} \;\; [\wedge I2].$$

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. $\square$

Slide 117

Example

Elimination and equivalence rules for disjunction

$$\frac{p \vee q \quad p \rightarrow r \quad q \rightarrow r}{r} \; [\vee E], \qquad \frac{p \vee q}{\neg(\neg p \wedge \neg q)} \; [\vee Equiv]$$

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.

The truth table formula

$(p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r) \rightarrow r$

is a tautology, and thus the theorem

$(p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r) \Rightarrow r$

is valid.

The equivalence rule $Equiv$ follows from De Morgan's laws. □

Slide 118

Finite State Machines

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 configuration of the system.

Slide 119

Now consider a simple model of a computer :

1. A computer is given information, the *input*.
2. At any instant, the computer contains information, stored in binary form, and this information defines the *state* of the computer. Since a computer has finite storage, it has a finite number of states.
3. The contents of some storage cells are available for *output*.
4. The computer operates in a deterministic manner because its actions are governed by a set of *rules*.

A computer has an internal clock that synchronizes its actions.

1. On a clock pulse, a new piece of input is read.
2. This changes some of its internal states.
3. The new states depend on the input and the current states.

Slide 120

Finite State Machines

Definition : A Finite State Machine.

A finite state machine (FSM) is defined by $M = (Q, A, B, F, G)$ where

1. Q is a finite set of states
2. A is a finite set of input symbols
3. B is a finite set of output symbols
4. F is a function that is defined by $F : Q \times A \rightarrow Q$
5. G is a function that is defined by $G : Q \times A \rightarrow B$

This is a general model of computation

Slide 121

States of a Vending Machine

- The response of a system to an input depends on previous inputs :

  Vending machine waiting (not in use) $\rightarrow$ insert money

  Money inserted $\rightarrow$ choose tea or coffee

  Vending machine waiting (not in use) and tea or coffee chosen $\rightarrow$ no change in current state

- The entire history of a vending machine is :

  insert money–choose coffee–insert money–choose tea–insert money–choose tea

  and the machine has only two states

Slide 122

State

- **State** – A representation of an indistinguishable input history, that is, given a current state of a machine, there are several inputs and previous states that could have generated the current state.
- The following input histories all result in the same state :

  money

  money–choose coffee–money

  money–choose coffee–money–choose tea–money

Slide 123

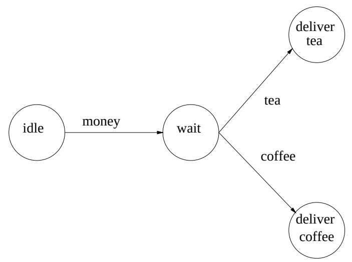
Modelling discrete systems :

- **State** System has a set of n states $Q = \{q_1, q_2, \cdots, q_n\}$
- **Input** Set of m discrete inputs from environment $A = \{a_1, a_2, \cdots, a_m\}$
- **Time** Discrete ticks of the clock, t

Slide 124

Example

Drinks Machine State Graph (1)



- State set $Q = \{$idle,wait,deliver coffee, deliver tea$\}$
- Input set $A = \{$money,tea,coffee$\}$ $\square$

Slide 125

State Transitions

- System is in state q_j and receives input a_k at time t_i
- The system jumps to the next state q_{j+1} at time t_{i+1}, dependent on the state and input at time t_i :

$$\text{state}\,(t_{i+1}) = f\,(\text{state}\,(t_i)\,,\text{input}\,(t_i))$$

- Define this by the next state function F :

$$F : Q \times A \to Q$$

- State at time t_{i+1} is given by $F(q, a)$.

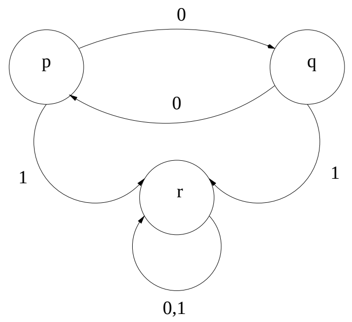
Slide 126

Example

- $Q = \{p, q, r\}$
- $A = \{0, 1\}$
- $F(p, 0) = q, F(q, 0) = p, F(r, 0) = r$
  $F(p, 1) = r, F(q, 1) = r, F(r, 1) = r$

Can display as either a table or a graph

Slide 127



The state graph

Slide 128

State	Input	Output
p	0	q
q	0	p
r	0	r
p	1	r
q	1	r
r	1	r

The tabular form

□

Slide 129

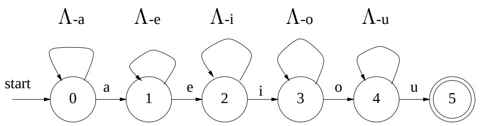
Summary

1. **Finite State Machine** (FSM) A computational model for dynamic events. It consists of a finite set of states, a finite set of inputs, and a state transition function that maps the current state and input to the next state
2. **State** The internal situation of the system that contains all the required information about previous events that is necessary for future operation
3. **State Table and State Graph** Representation of a FSM that gives a complete description of the state transition function

Slide 130

Example of Finite State Machines in Computer Science : Pattern Searching

Construct a finite state machine that examines a character string if a word has all the vowels in the correct order.



The symbol Λ denotes all the upper and lower case letters.

The shorthand $\Lambda - a$ denotes all these letters apart from the letter a.

Note : If we encounter a blank or any other invalid symbol, there is no transition, the processing stops and the input is *rejected.*

Slide 131

The Postfix Notation

- The argument q is written to the left of the function – qF_a
- Rewrite F as F_0 and F_1 :
  $pF_0 = F(p, 0) = q; qF_0 = F(q, 0) = p; rF_0 = F(r, 0) = r$
  $pF_1 = F(p, 1) = r; qF_1 = F(q, 1) = r; rF_1 = F(r, 1) = r$
- The state table and state graphs give a complete picture of F_a for all $a \in A$.

F_a is the **postfix notation** for the state transition function.

Note :

$F_a : Q \to Q$ where $a \in A$.

Slide 132

General State Table

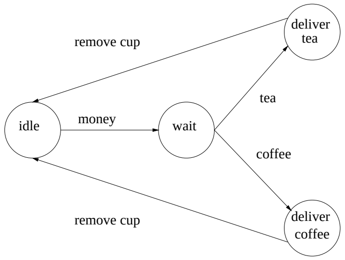
	State			
Input	q_1	q_2	$\cdots$	q_n
a_1	$q_1F_{a_1}$	$q_2F_{a_1}$	$\cdots$	$q_nF_{a_1}$
a_2	$q_1F_{a_2}$	$q_2F_{a_2}$	$\cdots$	$q_nF_{a_2}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
a_m	$q_1F_{a_m}$	$q_2F_{a_m}$	$\cdots$	$q_nF_{a_m}$

This is a correctly specified machine : F is specified for all $Q \times A$.

Slide 133

Consider again the drinks machine : Add the new input *Remove cup*

Drinks Machine State Graph (2)



Slide 134

Drinks Machine State Table (2)

	idle	wait	del. tea	del. coffee
money	wait	–	–	–
tea	–	del. tea	–	–
coffee	–	del. coffee	–	–
rem. cup	–	–	idle	idle

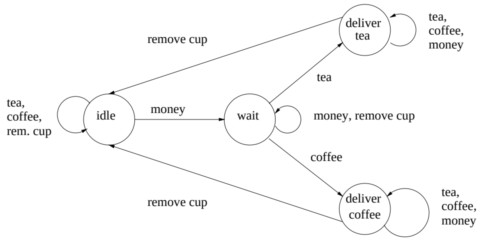
Note :

- Machine does nothing under certain circumstances
- The functions $F_{money}, F_{tea}, F_{coffee}$ and $F_{rem.cup}$ are partial functions from $Q \to Q$

Slide 135

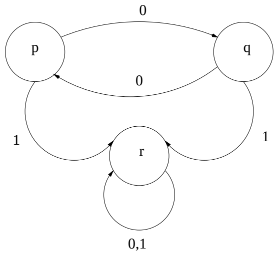
Drinks Machine State Graph (3)

Complete finite state machine - no partial functions



Slide 136

Consider again the machine



Start in state q and input successively : $0, 0, 1, 1, 0$

The output is : $q \xrightarrow{0} p \xrightarrow{0} q \xrightarrow{1} r \xrightarrow{1} r \xrightarrow{0} r$

Slide 137

- What is the output sequence due to the input

  0,0,0,0,0,0,0,0

  if the machine is started in state q ?
- What is the output sequence due to the input

  0,0,0,0,0,1,1,0

  if the machine is started in state r ?

Slide 138

Function induced by a sequence (1)

Recall that we defined the state transition function F by

$$F : Q \times A \rightarrow Q$$

and that the abbreviated notation $q_i F_{a_j}$ denotes $F(q_i, a_j)$.

What happens when the input consists of a sequence of characters (a string) ?

Advantage of postfix notation because it allows a concise form of the new state to be written.

Slide 139

Example

Let the machine be at state q and apply the input sequence $a_0 a_0 a_1 a_1 a_0$.

What is the mathematical expression that defines the new state of the system ?

1. The new state due to the input a_0 is $r_1 = qF_{a_0}$
2. The new state due to the input a_0 is
$$\begin{aligned} r_2 &= r_1 F_{a_0} \\ &= (qF_{a_0})\, F_{a_0} \end{aligned}$$
3. The new state due to the input a_1 is
$$\begin{aligned} r_3 &= r_2 F_{a_1} \\ &= ((qF_{a_0})\, F_{a_0})\, F_{a_1} \end{aligned}$$

Slide 140

4. The new state due to the input a_1 is
$$\begin{aligned} r_4 &= r_3 F_{a_1} \\ &= (((qF_{a_0})\, F_{a_0})\, F_{a_1})\, F_{a_1} \end{aligned}$$
5. The new state due to the input a_0 is
$$\begin{aligned} r_5 &= r_4 F_{a_0} \\ &= ((((qF_{a_0})\, F_{a_0})\, F_{a_1})\, F_{a_1})\, F_{a_0} \end{aligned}$$

□

Is there a more compact notation ?

Slide 141

Recall that A is set of inputs.

Definition The set A^* is the set of all finite strings from the set A, including the empty string Λ.

Example

If $A = \{a, b, c\}$, then

$$\begin{aligned} A^* &= \{\ldots \\ &\quad abc, acb, bac, bca, cab, cba, \\ &\quad ab, ba, ac, ca, bc, cb, \\ &\quad a, b, c, \\ &\quad \Lambda\} \end{aligned}$$

□

Slide 142

Recall previous example :

- Input string $x \in A^*$,

$$x = a_0 a_0 a_1 a_1 a_0$$

- New state :

$$qF_a = ((((qF_{a_0})\,F_{a_0})\,F_{a_1})\,F_{a_1})\,F_{a_0}$$

and

$$qF_\Lambda = q$$

Slide 143

Recursive Definition of State Transition Function

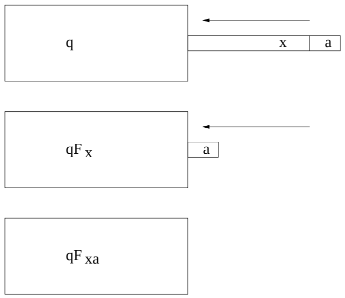
A string with at least one symbol can be written as a string x followed by the symbol a. Thus

- the effect of applying the string xa is the same as applying the string x followed by the symbol a

Let $x \in A^*$, $a \in A$ and $q \in Q$. Then

$$\begin{aligned} qF_\Lambda &= q \\ qF_{xa} &= (qF_x)\,F_a \end{aligned}$$

Slide 144



A diagrammatic interpretation of the state transition function

Slide 145

Example

Consider again the input sequence $a_0 a_0 a_1 a_1 a_0 \in A^*$.

$$\begin{aligned} qF_{00110} &= (qF_{0011})\, F_0 \\ &= ((qF_{001})\, F_1)\, F_0 \\ &= (((qF_{00})\, F_1)\, F_1)\, F_0 \\ &= ((((qF_0)\, F_0)\, F_1)\, F_1)\, F_0 \end{aligned}$$

$\square$

Slide 146

Summary

- **State Transition Function** $F_a : Q \to Q$ is the state transition function induced by the input symbol a; we write qF_a to show F_a applied to state q (postfix notation)
- **Partial Function** A state transition function that is not specified for all state–input combinations
- **Sequence of inputs** A sequence of inputs can be used to induce a state transition function by successively applying the state transition function that defines the FSM
- **Function induced by a sequence** If x is a sequence of input symbols, then F_x is a function that describes the overall state transition induced by x
- **Recursive Definition** The function F_x can be defined recursively on x by considering the addition of a symbol to the input sequence

Slide 147

Finite State Recognizers

A finite state machine may be used to *recognise* a specified input.

Examples During compilation, an input string is broken down into recognisable substrings, and these substrings are then treated as units in the next stage of the compilation process. The compiler should be able to recognise, for example, a sequence of input symbols that constitute a key word, or a reserved word in a programming language.

Objective : Introduction of a class of finite state *recognizers* that partition the set of all possible strings into a set of recognized strings and a set of rejected strings.

Slide 148

A proof about the state transition function

Given a finite state machine $M = (Q, A, F)$ with $q \in Q$ and $x, y \in A^*$, prove that $qF_{xy} = qF_xF_y = (qF_x)F_y$.

Method : Induction

1. If the string xy is of length $n = 1$, then $x = a$ and $y = \Lambda$ and the theorem is true (see earlier slide).
2. Assume that the theorem is true for a string of length n, and consider a string of length $n + 1$. Thus let $z = ya$ where a is a string of unit length, so that xz is a string of length $n + 1$.

Slide 149

$$
\begin{aligned}
qF_{xz} &= qF_{xya} && \text{Note 1} \\
&= (qF_{xy})\, F_a && \text{Note 2} \\
&= ((qF_x)\, F_y)\, F_a && \text{Note 3} \\
&= (qF_x)\, F_{ya} && \text{Note 4} \\
&= (qF_x)\, F_z
\end{aligned}
$$

3. Thus if the theorem is true for a string of length n, then it is true for a string of length $n + 1$. Since the theorem is true for a unit length string, it is true for a string of arbitrary length

□

Slide 150

Notes :

1. This follows because $z = ya$
2. This follows because a is a character (see earlier slide)
3. This follows by the inductive hypothesis – assume that the theorem is true for a string of length n
4. This follows because if $r = qF_x$, then

$$(rF_y)\, F_a = rF_{ya} = rF_z$$

Slide 151

Recognizers

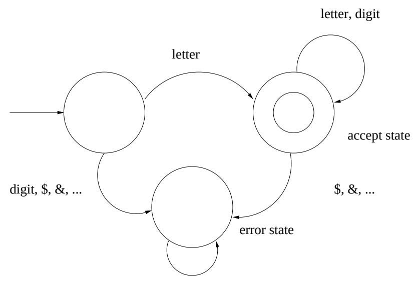
A finite state machine may be used as a *recognizer* to recognize or accept certain strings of symbols, and reject others. This arises because FSMs have a (limited) memory of previous inputs.

Example An interpreter or compiler for a programming language must determine if a string is a reserved word, an allowabe variable name, a real number etc. □

Consider

1. The state graph of the name of a variable
2. The state graph of a real number

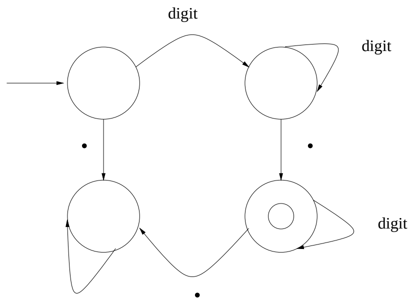
Slide 152



The state graph of the name of a variable.

Note : An accept state is shown as a double circle.

Slide 153



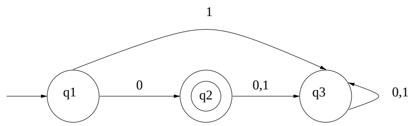
The state graph of a real number, which consists of a sequence of digits and '.' : $A = \{0, \ldots 9, .\}$.

Slide 154

- Each state of a recognizer M represents a class of *indistinguishable input histories*
- A recognizer splits the possible words $x \in A^*$ into two sets, a recognize (accept) state and a reject state
- We want to succintly describe the sets that are recognized by a particular finite state recognizer
- Previous examples :
  - The set of variable names
  - The set of real numbers

Slide 155

Example



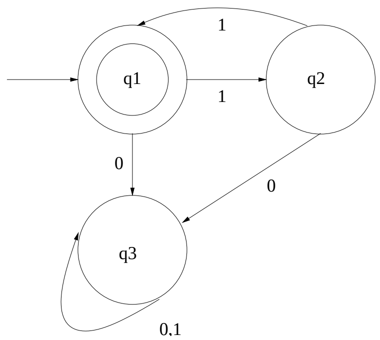
Accept set : $\{0\}$

All other sets are rejected

$\square$

Slide 156

Example



Accept Set : $\{\lambda, 11, 1111, 111111, \ldots\}$

Rejects all strings that do not consist of an even number of 1s. $\square$

Slide 157

Definition of a finite state recognizer

A **finite state recogniser** $M = (Q, A, F, i, T)$ is a finite state machine where

1. Q is a finite set of states
2. A is a finite set of input symbols
3. F is the state transition function
4. $i \in Q$ is the *initial state*
5. $T \subset Q$ is a set of one or more accept states

For a give input $x \in A^*$, M is said to recognize or accept x if $iF_x \in T$.

Slide 158

Summary

- **Finite State Recognizer** A finite state machine that is used to process strings of symbols with one state defined as the start state (shown graphically with an incoming arrow) and at least one accept state (shown as a double circle)
- **Accept State** A FS recognizer partitions the set of all possible input sequences into an accept state and a reject state

Slide 159

An Alphabet

Upto now, the input to a FSR has been a string of characters, with no constraints or rules. But

not all strings of characters can be recognised by a finite state recogniser

Questions :

1. What is the subset of all possible strings that can be recognised by a FSR ?
2. How can these strings be characterised ?

We will show that

Any string of characters that can be recognised by a FSR is regular, and any regular expression can be characterised by a FSR

Slide 160

Examples

1. Variables, constants and reserved words must be recognised by a FSR for compilation to occur
2. Suppose that you are searching the web and looking for references to the bird *robin*. Other web sites of interest could be found under *robins* and *Robin*. Instead of performing the search three times, can we use one search command for *robin, robins, Robin*, etc ?

□

A regular expression is used specifying text strings, such as occur in these examples.

Slide 161

Let A be a finite set; its elements are called **symbols** and A is called an **alphabet**. A^* denotes the set of all finite length **strings** or **words** over A. A^* can be defined recursively :

1. The **empty string** λ (the string with no symbols) belongs to A^*
2. Each member of A belongs to A^*
3. If x and y are strings in A^*, then so is $x \cdot y$, where $\cdot$ denotes concatenation or juxtaposition

Note : For any string x, $x \cdot \lambda = \lambda \cdot x = x$.

We can use induction to prove this definition of an alphabet.

Parts 1 and 2 are the base case, and part 3 is the inductive step.

Slide 162

Example

If $x = 1011$ and $y = 001$, then

$x \cdot y = 1011001, y \cdot x = 0011011$ and $y \cdot x \cdot \lambda x = 00110111011.$

□

Example

If $A = \{a, b\}$, then the alphabet consists of the characters a and b.

$abbaa$, $bbbbba$ and a are strings over A, that is

$A^* = \{\lambda, a, b, ab, ba, aaa, aab, aba, baa, \ldots, abbaa, bbbba, \ldots\}$

and $abbaa \cdot a$ yields the string $abbaaa$

□

The number of symbols in a string is called its **length** :

length (x) = 4, length (y) = 3, length ($x \cdot y$) = length ($y \cdot x$) = 7

Slide 163

Distinguish between

- the empty string λ :
  - the string with no characters
  - a type of *string* or *list of characters*
- the empty set $\emptyset$:
  - the set with no elements
  - a type of *set of string*

Notes :

1. If $A = \emptyset$, then $A^* = \{\lambda\}$
2. If A is not empty, then A^* is a denumerable (countably infinite) set

Slide 164

Note : A set S is countably infinite if there is a $1 : 1$ mapping between the natural numbers and the elements of S.

The integers, the rationals and the set of even numbers are countable sets, but the real numbers are not countable.

Example

If $A = \{a\}$, then

$A^* = \{\lambda, a, aa, aaa, aaaa, \cdots\}$

is an enumeration of A^*. □

Slide 165

Formal Languages

A *natural language* (English, etc.) must be distinguished from a *formal language*, which is used to model natural languages and to communicate with computers.

The rules of a natural language are complex and difficult to characterize completely, but the rules of a formal language can be completely specified.

Recall these definitions :

- For a natural language, A is the set of 26 characters, in both uppercase and lowercase, blanks and punctuation marks
- For a programming language, A is the same as that of a natural language, and the 10 digits $\{0, \cdots, 9\}$.
- A^* is the set of all finite length strings over A

Apply the definitions of A and A^* to a language.

A **formal language** over A is a subset of A^*.

Slide 166

Example

For a natural language, elements of the set A^* are

1. *on, cat, lay, the, bed, ' ', The, .*
2. *sausage, dog, steel*
3. *three, Tuesdays, ate I*

□

An English sentence can only be formed from (1) but not from (2). Item (3) is permissible in a formal language because it is *gramatically* correct; its *meaning* reveals that it is ridiculous.

Definition :

A **formal language** Σ over an alphabet A is a subset of A^*.

Slide 167

Example If $A = \{a, b\}$, then

$A^* = \{\lambda, a, b, aa, ab, bb, aaa, aab, aba, abb, baa, \cdots\}$

A few languages over A are

1. $\{\lambda, a, aa, aab\}$
2. $\{x \in A^* : |x| \leq 8\}$
3. $\{x \in A^* : |x|$ is odd $\}$
4. $\{x \in A^* : n_a(x) > n_b(x)\}$
5. $\{x \in A^* : |x| > 2$ and x 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. □

Slide 168

Methods for defining formal languages

- A formal language has been defined as a set of strings, called *words*
- But nothing has been stated about the type of words in the language

Example

The Pascal programming language, that is, the set of all syntactically correct Pascal programs, is a subset of all the strings that can be formed from the character set on a typical keyboard.

This definition is not helpful because it does not distinguish between a valid and an invalid Pascal program.

Solutions :

1. List all the possible Pascal programs
2. Define a set of rules for constructing a Pascal program, or more generally, a formal language. □

Slide 169

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

- Union
- Concatenation
- Kleene

Question : Why are these operations important ?

Answer : If a string is recognised by a finite state recogniser, then the string that results from the application of these operations *will also be recognised by the finite state recogniser*

Slide 170

Union operator

Let A be a finite set of symbols, and let $\Sigma_1, \Sigma_2 \subset A^*$ be two languages over A.

The *union* of Σ_1 and Σ_2, denoted by $\Sigma_1 \cup \Sigma_2$, is the set

$$\{x : x \in \Sigma_1 \text{ or } x \in \Sigma_2\}$$

which is the usual definition of the union of two sets.

Example

If $\Sigma_1 = \{10, 1\}$ and $\Sigma_2 = \{011, 11\}$, then

$\Sigma_1 \cup \Sigma_2 = \{10, 1, 011, 11\}$

□

Slide 171

Concatenation operator

Let A be a finite set of symbols, and let $\Sigma_1, \Sigma_2 \subset A^*$ be two languages over A.

The *concatenation* of Σ_1 and Σ_2, denoted by $\Sigma_1\Sigma_2$, is the set

$$\{xy : x \in \Sigma_1 \text{ and } y \in \Sigma_2\}$$

which is the set of all possible combinations of a string from Σ_1 followed by a string from Σ_2.

Example

If $\Sigma_1 = \{hope, fear\}$ and $\Sigma_2 = \{less, fully\}$, then

$\Sigma_1\Sigma_2 = \{hopeless, hopefully, fearless, fearfully\}$

□

Slide 172

Kleene (Closure) operator

Use exponential notation to indicate the number of items being concatenated

If A is an alphabet and $a \in A, x \in A^*$, and $\Sigma \subseteq A^*$, then

1. $a^k = aa \cdots a$
2. $x^k = xx \cdots x$
3. $A^k = AA \cdots A$
4. $\Sigma^k = \Sigma \cdots \Sigma$

where there are k factors in each product.

Slide 173

If $k = 0$, then

$$A = \{\lambda\}, \qquad \Sigma^0 = \{\lambda\}$$

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

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

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

□

Slide 174

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

$$\Sigma^0 = \lambda, \qquad \Sigma^{i+1} = \Sigma\Sigma^i$$

The *closure* Σ^* of Σ is defined as

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

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

$$\Sigma^* = \left\{\lambda, a, b, a^2, ab, ba, b^2, a^3, a^2b, aba, ab^2, ba^2, bab, b^2a, b^3, \cdots\right\}$$

□

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

□

Slide 175

Summary

If Σ_1 and Σ_2 are two languages over A that are recognised by a finite state recogniser, then

1. $\Sigma_1 \cup \Sigma_2$
2. $\Sigma_1\Sigma_2$
3. Σ_1^k and Σ_2^k

are sets whose elements are also recognised by a finite state recogniser.

These operations provide a set of rules that allow us to build up formal languages.

Slide 176

Regular Expressions

We have seen :

- The accept and reject states of a finite state machine are characterised by strings
- The characters in each string are drawn from an alphabet A
- The set of all words, or finite length strings drawn from A, is denoted by A^*
- A subset of A^* defines a language Σ
- The operations of concatenation, union and closure can be performed on the elements of Σ

Slide 177

A FSR defines a pattern - the set of all strings that label paths from an initial state to an accept state.

A string that can be described by a FSR is called a regular expression

- All regular expressions can be described by a FSR
- If a string can be recognised by a FSR, then it is a regular expression

Slide 178

Why are regular expressions important ?

A regular expression is the standard method for characterising text sequences in

- web search engines
- UNIX (vi, Perl, Emac, grep)
- WORD

A regular expression is an algebraic notation for the characterisation of a set of strings

Slide 179

How do we define a regular expression ?

In ordinary algebra, we distinguish between *operators* and *operands*.

Likewise, in the algebra of relational expressions, we also have operators and operands.

An operand is any one of the following :

1. A character
2. The character of zero length λ, corresponding to the set $\{\lambda\}$
3. The symbol $\emptyset$, the empty set $\{\}$
4. A regular expression

Only three operators are required to form a regular expression with these operands.

Slide 180

Let A be an alphabet; the regular expressions over A and the sets that they denote are defined recursively :

1. $\emptyset$ is a regular expression and denotes the empty set $\{\}$
2. λ (the empty string) is a regular expression and denotes the set $\{\lambda\}$, the set that only contains the string of zero length
3. For each $a \in A$, **a** is a regular expression and denotes the set $\{a\}$
4. If **r** and **s** are regular expressions denoting the sets R and S respectively, then **r**∪**s** (union), **rs** (concatenation) and **r*** (closure) are regular expressions that denote 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)

Slide 181

Example Let A be the alphabet $\{a, b, c\}$. The string

$$((\mathbf{a} \cup \mathbf{b})\mathbf{c})^*\mathbf{c}$$

is a regular expression consisting of the set of all strings that are formed by the union of a and b and the concatenation with c, and this is concatenated with c.

□

The string $\mathbf{s}^*$ is the regular expression that consists of *zero or more occurrences of s.*

The expression $\mathbf{a} \cup \mathbf{b} \cup$ is not regular because it cannot be formed from the primitive regular expressions.

Slide 182

Examples

- **00** is a regular expression of the set $\{00\}$
- **0**∪**1** represents $\{0, 1\}$
- **0*** represents $\{\lambda, 0, 00, 000, \cdots\}$
- (**0**∪**1**)* represents the set of all strings that contain 0s or 1s, including the empty string
- (**0**∪**1**)***00**(**0**∪**1**)* represents all strings with at least two consecutive 0s

□

Slide 183

Example

1. The string 10100010 is described by the regular expression **(0*10)***
2. The string 011100 is NOT described by the regular expression **(0∪(11)*)***
3. The string 000111100 is described by the regular expression **((011∪11)*(00)*)***

□

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 two (or more) ways :

(0∪1)* = [(0∪1*)*∪(01)*]*

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

Slide 184

Kleene's Theorem

Why are regular sets important ?

Kleene's theorem states that regular sets are important because these sets, and only these sets, can be recognised by a finite state machine.

Kleene's theorem :

Any set recognised by a finite–state machine is regular, and any regular set can be recognised by a finite state machine

Slide 185

Kleene's theorem states that only regular sets can be recognised by a finite–state recogniser.

But there are many sets that are not regular.

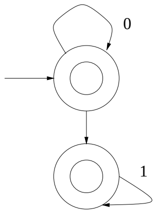
Example

1. The string 0^n1^n, consisting of n 0s followed by n 1s, is not regular
2. The string 0^*1^*, consisting of an arbitrary number of 0s followed by an arbitrary number of 1s, is regular

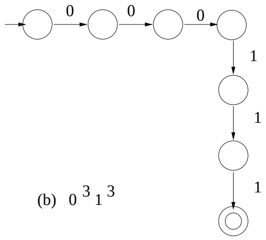
By Kleene's theorem, 1 cannot be recognised by a finite–state recogniser, but 2 can be recognised by a finite–state recogniser.

Thus finite–state machines are not a complete model of computation.

Slide 186



(a) 0^*1^*

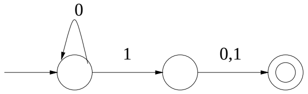


(b) $0^3 1^3$

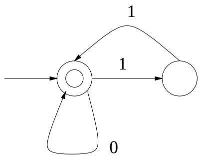
A different finite–state machine is required in (b) for different values of n

Slide 187

Examples of finite state machines for regular expressions



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



$(0 \cup (11)^*)^* = (0 \cup (11))^*$

Slide 188

Summary

- **Regular Set** A set that is accepted by a finite state recogniser using the operations of concatenation, union and closure
- **Regular Expression** Compact notation for representing regular sets, built up by using the empty string, the symbols of the input alphabet and the operations of concatenation, union and closure

Slide 189

Regular Expressions in Action

1. `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
2. 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 a FSM of a token is obtained, using a regular expression

Slide 190

Examples of `egrep`

- A string s matches a regular expression r if s is in the set of strings generated 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
```

□

Slide 191

Examples of `egrep`

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

Example

(**0**∪**1**)***00**(**0**∪**1**)* is denoted by (0|1)*00(0|1)*

□

The argument in the `egrep` command is a regular expression. By Kleene's theorem, a FSM that represents this expression can be built, and a software model of this FSM is developed.

Slide 192

`egrep` syntax

- **Alphabet** The period character . matches any character in the alphabet A. The set A^* is matched by $.^*$
- **Empty String** `egrep` cannot express the empty string λ, 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 $\$$

Slide 193

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
```

Slide 194

- 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
```


Slide 195

- Crossword puzzle : "s-blank-blank-u-blank-t-blank-blank-e"

```
egrep '^s..u.t..e$' /usr/dict/words
seductive
structure
```

□

Slide 196

Finite State Machines with Output

Summary :

Finite state machines provide a simple mathematical model of a computer :

- Reading the letters of the input string is equivalent to executing instructions because it causes the state of the machine to change, which causes
  1. the contents of memory, and
  2. the control section

  to change.

But what about output ?

Slide 197

Example Consider again the example of the vending machine. At various stages during the operation of the machine, it may output prompts for more coins or more option buttons to be pressed.

We are concerned with both an **input** sequence and an **output** to define fully a vending machine.

□

Extend finite state machines to Mealy or Moore machines to represent input–output systems.

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

Slide 198

Definition of a Mealy machine :

A Mealy machine consists of a 5–tuple $M = (S, A, B, F, G)$ where

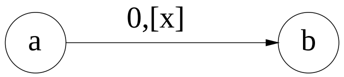
- S is a finite set of states
- A is a finite set of input symbols
- B is a finite set of output symbols
- $F : S \times A \to S$ is the state transition function
- $G : S \times A \to B$ is the output function

Given a system in state $s \in S$ with an incoming input $a \in A$, then at the next time instant, the state changes to $F(s, a) \in S$ and an output $G(s, a) \in B$ is generated.

Slide 199

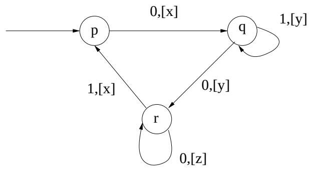
Example Let $S = \{p, q, r\}$, $A = \{0, 1\}$, $B = \{x, y, z\}$

	input	p	q	r
F	0	q	r	r
F	1	–	q	p
G	0	x	y	z
G	1	–	y	x



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

Slide 200



Note : There is no arrow labelled by 1 from state p, and thus the output when the machine is in state p and receives an input of 1 is not defined.

Slide 201

Mealy Machine – Notation

For notational convenience, we replace the set of state transition functions

$$F_a : S \to S \qquad (a \in A)$$

by

$$sF_a = F(s,a) \qquad \text{for all } \; s \in S$$

Similarly, we replace the set of output functions

$$G_a : S \to B \qquad (a \in A)$$

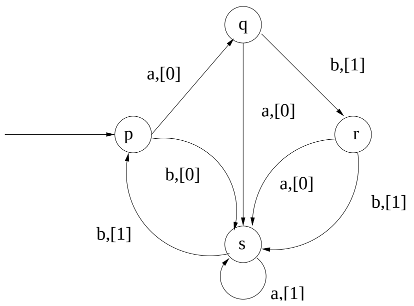
by

$$sG_a = G(s,a) \qquad \text{for all } \; s \in S$$

Slide 202

Example

What is the output of the Mealy machine due to the input sequence $ababb$?



Slide 203

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.

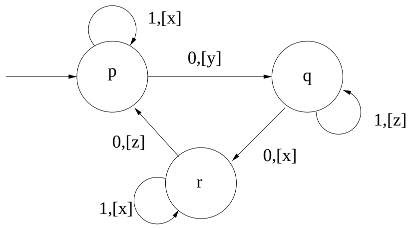
Note : The length of the output string is equal to the length of the input string (=5).

□

Slide 204

Example

What is the output of the following machine due to the input sequence 101 ?



Answer : xyz

Slide 205

Minimisation 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 minimise a finite state machine :

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

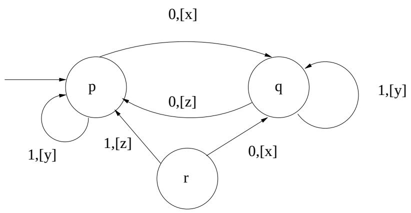
1. Given an input string x, M and M' produce the same output v from their starting states
2. M' has fewer states than M. If M' does not exist, then M is already minimal

Two stages :

1. Eliminate *unreachable* states
2. Merge *equivalent* states

Slide 206

Stage 1 : Eliminate *unreachable* states.



The state r is unreachable :

For every string x, the state r is never reached

Slide 207

Example

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

	input	p	q	r	s	t
F	0	q	t	r	s	p
F	1	t	q	r	q	p
G	0	x	y	y	x	y
G	1	x	y	y	x	y

Slide 208

1. Apply input 1 at state p and reach state t
2. State t leads to state p for 0 and 1. Hence only consider input 0 to state p
3. Apply input 0 at state p and reach state q
4. Reach either state t for input 0 (considered above) or state q for input 1

Thus states r and s are unreachable and can be deleted

$\square$

Slide 209

Stage 2 : Merge *equivalent* states

Definition : Two states p and q in a Mealy machine are equivalent if they yield the same output for all $x \in A^*$.

- Minimising M corresponds to finding the sets of equivalent states
- Impossible to find the equivalent states directly since there is no limitation on the length of the input string x
- Consider k–**Equivalent States**

Note : The notion of equivalent states considers a finite state machine as a 'black box'; we are only concerned with the input–output relationship, not the path traversed by the input stream through the states to achieve the output.

Slide 210

Definition : Two states p and q are k–**equivalent** if they yield the same output for for all $x \in A^*$, $|x| \leq k$.

Definition : A set of equivalent states is called an equivalence class.

If M can be minimised to a minimal Mealy machine M', then M' has states that are equivalence classes of M.

Method :

1. Find 1–equivalent states, that is, states that produce the same output for 1–length inputs
2. Use these states to find 2–equivalent states
3. Use these states to find 3–equivalent states, etc

Slide 211

1. All states are 0–equivalent because they all yield the same output (the zero output) for an input string of length zero (the empty string).
2. The 1–equivalent classes are obtained by examining those states that produce the same output for an input string of length one.
3. The 2–equivalent classes are obtained by examining those states that produce the same output for an input string of length two.

Equivalence classes have the following characteristics :

- All states in the same class have the same output.
- For each input symbol, all states in the same class proceed to states that are all in the same class.

Slide 212

A *necessary* but not *sufficient* for a 2–equivalent state is that it be a 1–equivalent state.

Thus : Consider all the 1–equivalent classes to calculate the 2–equivalent classes, and use recursion. More generally

1. We know how to find 1–equivalent states
2. Suppose we know which states are $k-1$ equivalent
3. If states p and q are k–equivalent, they must produce the same output strings for all input strings of length no greater than k, and in particular, for all input strings of length no greater than $k-1$
4. Thus p and q must be at least $k-1$ equivalent

Thus, to find k–equivalent states, look for $k-1$ equivalent states whose next states are $k-1$ equivalent for all input symbols

Slide 213

Example

state	p	q	r	s	t	u
F_0	q	p	t	u	r	q
F_1	r	t	s	r	s	r
G_0	x	y	y	x	y	y
G_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–equivalent classes of M are

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

Slide 214

Finding the 2–equivalent classes : Consider each of the 1–equivalent classes separately.

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

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

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

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

Slide 215

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

Since t and r are 1–equivalent, and s is obviously 1–equivalent, it follows that $\{r, t\}$ is a 2–equivalent class.

Thus the 2–equivalent classes are

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

Slide 216

Finding 3–equivalent classes :

- p and s are not 3–equivalent because q and u are not 2–equivalent
- r and t are 3–equivalent because r and t are 2–equivalent, and s is 2–equivalent

Thus the 3–equivalent classes are

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

Slide 217

Finding 4–equivalent states :

Since $\{r,t\}$ is a 3–equivalent class, the 3– and 4–equivalent classes are identical, and no further refinement is possible.

The new Mealy machine is obtained by defining a new state $w = r = t$:

state	p	q	w	s	u
F_0	q	p	w	u	q
F_1	w	w	s	w	w
G_0	x	y	y	x	y
G_1	y	x	y	y	x

$\square$

Slide 218

Example

state	p	q	r	s	t	u	v
F_0	r	s	p	q	v	r	t
F_1	s	r	t	u	u	p	p
G_0	x	y	x	y	y	x	y
G_1	x	y	x	y	y	x	y

The 1–equivalent classes are

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

Slide 219

Calculate the 2–equivalent classes. Consider each of the 1–equivalent classes separately.

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

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

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

Since s and p are NOT 1–equivalent, p and u are not 2–equivalent. Thus, the 2–equivalent classes from the 1–equivalent class $\{p,r,u\}$ are $\{p,r\}$ and $\{u\}$.

Slide 220

Consider now the the 1–equivalent class $\{q,s,t,v\}$.

- States q and s move to states s and q for an i/p of 0
- States q and s move to states r and u for an i/p of 1

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

- States q and v move to states s and t for an i/p of 0
- States q and v move to states r and p for an i/p 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–equivalent class.

Thus the 2–equivalent classes are

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

Slide 221

To calculate the 3–equivalent classes, consider the 2–equivalent states.

- $\{p, r\}$ is 3–equivalent.
- $\{u\}$ is 3–equivalent.
- $\{q, s\}$ is not 3–equivalent.
- $\{q, t\}$ is not 3–equivalent.
- $\{q, v\}$ is 3–equivalent.
- $\{s, t\}$ is 3–equivalent.

Thus the 3–equivalent classes are

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

Exercise : Show that the 4–equivalence classes are the same as the 3–equivalence classes, and thus no further reduction is possible.

Slide 222

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

state	a	b	c	u
F_0	a	c	b	a
F_1	c	a	u	a
G_0	x	y	y	x
G_1	x	y	y	x

□

Slide 223

Summary : Minimisation algorithm

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

Let $\{p_1, p_2, \ldots, p_n\}$ be N–equivalent–states. If these n states are merged into one state, then

- The system will produce exactly the same output for all inputs of length less than or equal to N.
- The original machine and the reduced machine may produce different outputs for strings of length greater than N.

Slide 224

Languages and Grammars

- A regular expression defines a pattern according to a specified set of rules
- Another method of defining a pattern, based on a grammar, is now described

Consider the English language, whose alphabet is

$$A = \{a, b, c, \ldots, x, y, z, \text{punctuation marks}\}$$

The language of A is

$$\Sigma = \{\text{words in a dictionary, punctuation marks}\}$$

The rules of grammar define the combinations and arrangements of words that yield valid sentences.

Abstract languages may be constructed in one of two ways :

1. An alphabet and an exhaustive list of valid words
2. An alphabet and a set of rules that define acceptable words

Slide 225

- Grammars are required for the study of programming languages
- Grammars provide a succinct notation for describing the syntax of typical programming languages
- A grammar can be used to generate a parser (important in compilation) for a programming language

The language that is generated by a regular grammar is a set of regular expressions, which can therefore be implemented by a finite state machine

Use the grammar of a natural language to consider the grammar of a programming language.

Slide 226

A natural language has a grammar that defines the construction of valid sentences. Is this enough ?

- *Birds sing* is grammatically correct and meaningful
- *Wednesday sings* is grammatically correct but meaningless

Distinguish between the *semantics* and *syntax*

- The rules of grammar are called the syntax of a language
- The meaning of words is called the semantics of a language

In general, computer languages are syntactic and not semantic.

Example

1. In order to parse the sentence *Birds sing*, it is necessary to distinguish between a noun and a verb
2. In order to parse the arithmetic expression $(3 + 5) \times 6$, it is not necessary to know the characteristics of the numbers 3, 5 and 6

Slide 227

Consider the English sentence

The black cat sat on the mat

Let w be the string that contains these words :

$$w = \{\text{the, black, cat, sat, on, the, mat}\}$$

Thus

$$\begin{aligned} w^* \quad &= \quad \{\Lambda, \text{The}, \cdots, \text{The black cat sat on the mat}, \\ &\quad\quad \text{The black cat sat sat on the mat}, \cdots\} \end{aligned}$$

Partition w^* into two sets : correct and incorrect

Question : How can sentences be constructed from w ?

Slide 228

Break the sentence into two :

1. a *subject phrase* that states who or what is doing something
2. a *predicate* that states what is being done

subject phrase : the black cat

predicate : sat on the mat

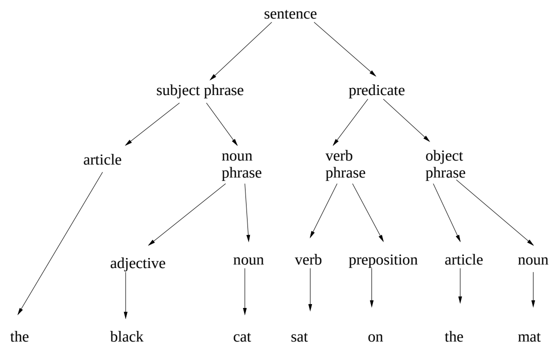
Break these parts up again :

1. The subject phrase is split into an *article* (the) and a *noun phrase* (black cat)
2. The predicate is split into a *verb phrase* (sat on) and an *object phrase* (the mat)

Break up again :

1. The noun phrase is split into an *adjective* (black) and a *noun* (cat)
2. The verb phrase is split into a *verb* (sat) and an *preposition* (on)

Slide 229



The complete decomposition tree

Slide 230

The branches of the decomposition tree yield *production rules*, which tell us how to break down a sentence into specific parts :

sentence	$\rightarrow$	*subject phrase* followed by *predicate*
subject phrase	$\rightarrow$	*noun phrase* followed by *article*
predicate	$\rightarrow$	*verb phrase* followed by *object phrase*
noun phrase	$\rightarrow$	*adjective* followed by *noun*
verb phrase	$\rightarrow$	*verb* followed by *preposition*
object phrase	$\rightarrow$	*article* followed by *noun*

Since the semantics of a sentence are not considered, the sentence

Itchy the apple eats a jumpy jumpy jumpy bear

can be decomposed.

Slide 231

Then we make substitutions from the dictionary to yield specific sentnces :

art	$\rightarrow$	'the'
adj	$\rightarrow$	'black'
adj	$\rightarrow$	'white'
noun	$\rightarrow$	'cat'
noun	$\rightarrow$	'mat'
prep	$\rightarrow$	'on'
verb	$\rightarrow$	'sat'

These production rules generate other gramatically correct sentences, eg.,

1. A white cat sat on the black mat
2. A mat sat on the white cat

Slide 232

The rules of grammar are really suggestions of possible substitutions.

How can this procedure be defined formally ?

1. We have a set of primitive language concepts, called the *variables* of the grammar :

   $V = \{$sentence, subject phrase, predicate, article, ... etc.$\}$
2. We have a set of words from a dictionary, called the *terminals* of the grammar :

   $T = \{$'a', 'the', 'cat', 'sat','on', ... etc.$\}$

Note that $V \cap T = \emptyset$

Slide 233

1. The construction of the leaves of the tree is controlled by the production rules F, which are function that map symbols from $(V \cup T)^*$ to $(V \cup T)^*$
2. A starting point for the construction of the root of the tree is required. This is an element of V. For the example, it is the sentence S

Thus a grammar contains :

1. a set of variables V
2. a set of terminators T
3. a set of production rules F
4. a starting point S

Slide 234

Definition : A *grammar* is a 4–tuple $G = (V, T, F, S)$, where

1. V is a finite non–empty set (the variables)
2. T is a finite non–empty set (the terminators)
3. F is a finite set of ordered pairs (p, q) where $p, q \in (V \cup T)^*$ and p contains at least one symbol from V
4. $S \in V$ is the start symbol

Slide 235

Example Consider the grammar with

1. variables $V = \{A, B\}$
2. terminators $T = \{a, b\}$
3. starting symbol S
4. production rules

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

The words generated by this grammar includes

$aa, aaab, aaaaaab$

□

Slide 236

Example Consider the grammar with

1. variables $V = S$
2. terminators $T = \{a\}$
3. starting symbol S
4. production rules

$$S \to aS, \qquad S \to \Lambda$$

Slide 237

Apply the first production rule six times, and then the second production rule :

$$\begin{aligned} S &\rightarrow aS \\ &\rightarrow aaS \\ &\rightarrow aaaS \\ &\rightarrow aaaaaS \\ &\rightarrow aaaaaaS \\ &\rightarrow aaaaaa\Lambda \\ &= aaaaaa \end{aligned}$$

□

Slide 238

What is Λ ?

- It is not a variable since a production rule of the form $\Lambda \rightarrow$ *something* does not exist
- It is not a terminator since $\Lambda aa\Lambda = aa$
- The production rule $N \rightarrow \Lambda$ deletes the variable N from any place in a working string

Thus Λ is neither a variable nor a terminator, but has a special status of it sown.

Slide 239

Generation

Example Consider the grammar $G = (V, T, F, X)$ where $V = \{X\}, T = \{a\}$ and the production rules are

$$X \rightarrow aXb, \qquad X \rightarrow \Lambda$$

Thus

$$X \rightarrow aXb \rightarrow aaXbb \rightarrow aaaXbbb$$

etc. □

Use the notation $X \rightarrow^*$ to indicate that there exists a sequence of direct generations from X to $aaaXbbb$.

$$X \rightarrow^* aaaXbbb$$

- use $\rightarrow$ in the production rules
- use $\rightarrow^*$ in the unfinished stages during the generation of the word

Slide 240

Can this procedure be generalised to yield an alternative definition of a grammar ?

Definition : Consider the grammar $G = (V, T, F, S)$. If $x, y \in (V \cup T)^*$, then $x \rightarrow^* y$ if and only if there exist $w_1, w_2, \ldots w_n \in (V \cup T)^*$ such that

1. $x = w_1$
2. $y = w_n$
3. $w_i \rightarrow w_{i+1}, 1 \leq i \leq n-1$

We say that x *generates* y in G, y is *generated* from x in G.

Slide 241

Example Consider the grammar $G = (V, T, F, X)$ where

$$V = \{X\}, \qquad T = \{a, b, c\}, \qquad F = \{X \to aXb, X \to c\}$$

In general we have

$$X \to aXb \to aaXbb \to \cdots \to a^n cb^n$$

Thus $X \to^* a^n cb^n$ and $a^n cb^n \in T^*$.

These are the only words in T^*.

□

Slide 242

The Language of a Grammar

Definition : The language $L(G)$ that is generated by a grammar G is the set of all strings of terminals that can be produced from the starting point S using the production rules as substitutions.

Example Consider the grammar $G = (V, T, F, X)$ where

$$V = \{X\}, \qquad T = \{a, b, c\}, \qquad F = \{X \to aXb, X \to c\}$$

The set $L(G) = \{a^n cb^n | n > 0\}$ is called the *language* of the grammar G. □

1. $L(G)$ is the set of all terminal words that can be generated from the start symbol S
2. Once a string of terminals has been obtained, no production rules can be applied, and thus the string cannot generate any more words

Slide 243

Definition : Context–free grammar

Thus far, only a special class of grammars, called *context–free grammars* have been considered. They are characterised by the form of their production rules :

The left hand side of the production rule of a context–free grammar consists only of a single variable.

Definition : Consider the grammar $G = (V, T, F, S)$.

Then G is called a *context–free grammar* if and only if each production rule $p \to q$ is of the form $v \to w$, where $v \in V$ and $w \in (V \cup T)^*$.

This grammar is called context–free because v is replaced by w *wherever* v appears.

Slide 244

By placing restrictions on the type of production rules that are allowed, different grammars are generated.

A *regular grammar* is a particular type of context-free grammar

Definition : Consider the grammar $G = (V, T, F, S)$. Then G is called a *regular grammar* if the production rules are of the form

$$\begin{aligned} Y &\to P, \qquad Y \in V, \qquad P \in T^* \\ Y &\to PZ, \qquad Y, Z \in V, \qquad P \in T^* \end{aligned}$$

Slide 245

Regular Grammars and Context–Free Grammars

- Every regular grammar is a context–free grammar but the converse is not true; a context–free grammar is not necessarily a regular grammar.
- Regular grammars are simpler than context–free grammars, and can therefore be processed more quickly. This affects, for example, compilation.
- Context–free grammars are more expressive (powerful). For example, nested brackets can only be captured by a context–free grammar, not a regular grammar. This affects the compilation of arithmetic expressions.
- There exist context–free languages that are generated by a regular expression, and there exist context–free languages that are not generated by a regular expression.
- The context–free languages that are not generated by a regular expression are used, typically, for describing and processing natural languages. These languages are recognised by push down automata.

Slide 246

Example Consider two grammars, G_1 and G_2 where $T = \{a, b\}$ and $S, A, B \in V$, have the following productions

1. G_1 :
$$S \to aA, \qquad A \to aAB, \qquad B \to b, \qquad A \to a$$
2. G_2 :
$$S \to aB, \qquad B \to bA, \qquad B \to b, \qquad B \to a, \qquad A \to aB, \qquad A \to a$$

G_1 is a context–free grammar

G_2 is a regular grammar

□

Slide 247

Example Consider a grammar G where $T = \{a, b\}$ and $S, A, B \in V$, have the following productions

$$S \to aB, \qquad B \to bA, \qquad B \to b, \qquad A \to aB$$

This is a regular grammar, and the language of the grammar is

$$L(G) = \{(ab)^n = ababab \cdots ab : n \in N\}$$

□

Slide 248

Languages derived from grammars are called *formal languages*

- If the grammar is defined first, then the language will follow as a consequence of the definition
- If the language is specified first as a well–defined set of strings, then we may seek a grammar that defines it

Slide 249

Example Consider the following two languages on the letters a and b :

$$L = \{a^r b^s : r, s > 0\} \qquad L' = \{a^n b^n : n > 0\}$$

Both languages consist of words that start with a and end with b, but the number of a's and b's in L' must be equal.

1. A regular grammar that generates L is

$$\begin{aligned} S &\rightarrow aA, \quad A \rightarrow aA, \quad A \rightarrow b \\ A &\rightarrow bB, \quad B \rightarrow bB, \quad B \rightarrow b \end{aligned}$$

2. A context–free grammar that generates L' is

$$S \rightarrow ab, \qquad S \rightarrow aSb$$

□