CAB203 Discrete Structures

Lecture Notes

Jack Girard 2023

Contents

1 What is Mathematics?

1.1 What is Mathematics?

1.1.1 Abstraction

Abstraction can be used to simplify a problem by ignoring all the information that is not needed. They capture the relevant properties of a situation, and the relationships between them. Those properties can then be used to work out the solution.

- You have x apples
- You have y friends
- Are there enough apples for all your friends?

The usual abstraction for problems involving counting is *natural numbers* (non-negative integers). By abstracting away irrelevant properties such as size, shape, colour etc. the problem can be simplified to:

$$x \geq y?$$

Limitations of abstractions can include:

- Not enough information
- Too much information
- Incorrect information

1.1.2 Mathematical Theories

An abstraction without reference to a particular problem is called a *mathematical theory*, usually consisting of:

- Mathematical objects (numbers, operations, etc.)
- Axioms (statements about how objects relate to each other)

1.1.3 Axioms

Some examples of axioms for the natural numbers include:

1. 0 is a natural number
2. $x = x$
3. if $x = y$ then $y = x$
4. if $x = y$ and $y = z$ then $x = z$
5. if $x = w$ then w is a natural number
6. $S(x)$ is a natural number
7. if $S(x) = S(y)$ then $x = y$
8. $S(x) = 0$ is always false

Axioms can be combined to create new true statements. For example, axiom 6 in the list above states $S(x)$ is a natural number, so by combining it with itself it can be deduced that $S(S(x))$ is also a natural number.

Note

S refers to the *successor function* that increments a natural number.

$$S(x) = x + 1$$

1.1.4 Mathematical Objects

Mathematical objects are abstract objects that can correspond to concrete objects. They can only be defined in how they relate to other objects - this is done through axioms. Formally, objects are just symbols. They have no meaning, and are just names.

Relationships between objects are given by *propositions*. Propositions are statements that can be true or false. Axioms are propositions that assert to be true for objects in the mathematical theory.

1.1.5 Models

A mathematical theory can apply to a real situation if:

- Every object in the theory matches up to something in the real situation (at least hypothetically)
- All axioms in the theory remain true in the real situation

If this can be done, the real situation can be defined as a *model* for the theory.

Another instance of a model is a *mathematical model*. This type of model is an abstraction of a particular system to be studied and analysed in a mathematical way, and is the primary focus of this course when referring to "model".

1.1.6 Truth in Mathematics

Statements in mathematics are always relative to a particular mathematical theory. A statement may be true in one theory and false in another.

- For example, $ab = ba$ is true for real numbers, but not for matrices.

A true statement in a theory is irrelevant to a real situation, **unless** it is a model for that theory. The rules of logic guarantee that true statements in a theory are also true in every model of the theory.

Note

It is possible for every statement in a theory to be true **and** false. This is called an *inconsistent* theory and cannot have any models.

1.2 Modular Arithmetic

1.2.1 Mathematical Definitions

A mathematical definition creates a short name for some concept. This is purely for brevity and does not express new things. In definitions, italics are used to emphasise the words that are being defined.

1.2.2 "Divides"

A mathematical definition for "divides" is as follows:

Divides Definition

Let a, b be integers. If there is another integer c such that $ac = b$, then it can be said that a *divides* b, written as $a|b$. Equivalently, it can also be said that b is *divisible* by a.

This definition states all the following are true:

- $a|b$
- a divides b
- b is divisible by a
- There is some integer c such that $ac = b$

1.2.3 Modular Arithmetic and Equivalence

For any positive integer n, it can have arithmetic modulo n. Modular arithmetic replaces equality with *modular equivalence*. a mathematical definition for modular equivalence is as follows:

Modular Equivalence Definition

if $n|(a-b)$, then a and b are *equivalent modulo* n, written as

$$a \equiv b \pmod{n}$$

Modular equivalence carries similar properties to addition, subtraction, and multiplication. If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then:

- $a + c \equiv b + d \pmod{n}$
- $a - c \equiv b - d \pmod{n}$
- $ac \equiv bd \pmod{n}$

1.2.4 Mod Operator

The mod operator is used to perform modular arithmetic. A mathematical definition for the operator is as follows:

Mod Operator Definition

$a \bmod n$ is the smallest non-negative b such that $a \equiv b \pmod{n}$

Equivalently, $a \bmod n$ is the remainder you get when you divide a by n. In most programming languages, the mod operator is denoted by %.

1.2.5 Example: Proving Lemma

Lemma

Let a and b be integers. If $a \bmod b = 0$, then $b|a$.

This lemma can be proven using previously stated definitions:

$$a \bmod b = 0 \text{ is the same as } a \equiv 0 \pmod{b}.$$
$$a \equiv 0 \pmod{b} \text{ is the same as } b|(a-0).$$
$$a - 0 = a \text{ therefore } b|a.$$

This example is often used in programming to determine divisibility, or test if a number is even.

1.3 Exponents and Logarithms

1.3.1 Exponents

Exponentiation refers to multiplying a base b by itself n number of times.

$$b^x = \underbrace{b \times \cdots \times b}_{x \text{ times}}$$

- b is called the *base*
- n is called the *exponent*

1.3.2 Laws of Exponents

Some examples of laws regarding exponents are:

- $(ab)^n = a^n \cdot b^n$
- $a^m \cdot a^n = a^{m+n}$
- $a^{m-n} = \frac{a^m}{a^n}$ (when $a \neq 0$)
- $a^{-n} = \frac{1}{a^n}$ (when $a \neq 0$)
- $a^0 = 1$
- $(a^m)^n = a^{m \cdot n}$

1.3.3 Exponents in Computer Science

Bases of 2 are very common in computer science due to computers working on bits at the fundamental level, which only involves two states. Numbers involved in counting bits can become very large, as such there are prefixes to refer to quantities of bits, similar to SI unit prefixes:

- *Kilo-* is to multiply by 2^{10}
- *Mega-* is to multiply by $2^{20} = (2^{10})^2$
- *Giga-* is to multiply by $2^{30} = (2^{10})^3$
- *Tera-* is to multiply by $2^{40} = (2^{10})^4$
- *Peta-* is to multiply by $2^{50} = (2^{10})^5$
- *Exa-* is to multiply by $2^{60} = (2^{10})^6$

For example, 1 kilobit = 2^{10} = 1024 bits.

1.3.4 Logarithms

Logarithms are the inverse of exponents. That is to say: *the power to which a number must be raised in order to get some other number.*

$$\log_b n = x$$

Which is equivalent to the exponential $b^x = n$.
For example:

$$2^x = 1024 \tag{1}$$
$$\log_2 1024 = x \tag{2}$$
$$x = 10 \tag{3}$$

1.3.5 Laws of Logarithms

Some examples of laws regarding logarithms are:

- $\log_a 1 = 0$
- $\log_a a = 1$
- $\log_a(x \cdot y) = \log_a x + \log_a y$
- $\log_a x^y = y \log_a x$
- $\log_a \frac{1}{y} = -\log_a y$
- $\log_a \frac{x}{y} = \log_a x - \log_a y$
- $\log_b x = (\log_b a) \cdot \log_a x$

1.3.6 Base Transformation Law

Logarithm base 10, known as the *common logarithm* is the common form of logarithms, however computer science often uses logarithm base 2. It is possible to calculate base 2 from base 10 using base transformation:

$$\log_a x = \frac{log_b x}{\log_b a}$$

1.3.7 Floor and Ceiling Functions

If a decimal result is undesirable, floor and ceiling functions can be used to round down or round up respectively.

- $\lfloor a \rfloor$ means to round down to the next integer below a (**floor**)
- $\lceil a \rceil$ means to round up to the next integer above a (**ceiling**)

Example:

$$\begin{aligned} \log_2 3 &= 1.5849625007 \\ \lfloor \log_2 3 \rfloor &= 1 \\ \lceil \log_2 3 \rceil &= 2 \end{aligned}$$

2 Data Representation

2.1 Bits and Bit Strings

2.1.1 Bits

A *bit* is a fundamental unit of information that has 2 states, usually labelled 0 and 1. Everything that happens in a computer involves bits, whether that's inputting, storing, manipulating, or outputting data.

2.1.2 Bit Strings

Bit strings are a sequences of n number of bits. A single bit can also be classified as a bit string.

- 0
- 00000
- 10101110001
- 11111111111111111111111111111111111

2.1.3 Bit String Notation

Some common notation regarding bit strings is as follows:

- Overline over variables to indicate a bit string: $\overline{x}$
- The set of all strings of length n is $\{0,1\}^n$ (n-bit strings)
- All bit strings of all lengths are members of $\{0,1\}^*$
- The jth bit in $\overline{x}$ is $\overline{x}_j$ (j goes from 0 to $n-1$)

Some extra things to note:

- Bit strings are counted right-to-left. $\overline{x}_0$ is the rightmost bit
- For each $\overline{x}_j$ there are two possible values, 0 and 1
  - This means there are 2^n possible bit strings of length n

2.1.4 Bit Operations

Bitwise operations can be performed on bit strings of the same length. There are two types of bit operations:

- Single bit or bit pair operations
- Bit string operations

The four typical bitwise operations performed are **NOT**, **AND**, **OR**, and **XOR**.

2.1.5 Bitwise NOT

NOT is a unary operator that flips a bit. 0 becomes 1, and 1 becomes 0.

Table 1: NOT Truth Table

x	$\neg x$
0	1
1	0

Note

In programming, the operator for NOT is usually $\sim$

2.1.6 Bitwise AND

AND is a binary operator that results in 1 if **both** operands are also 1, otherwise it will result in 0.

Table 2: AND Truth Table

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

Note

In programming, the operator for AND is usually &

2.1.7 Bitwise OR

OR is a binary operator that results in 1 if **either** operands are also 1, otherwise it will result in 0.

Table 3: OR Truth Table

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Note

In programming, the operator for OR is usually |

2.1.8 Bitwise XOR

XOR is a binary operator that results in 1 if both operands are **different** values, otherwise it will result in 0.

Table 4: XOR Truth Table

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Note

In programming, the operator for XOR is usually ^

2.1.9 Concatenation

Bit strings can be concatenated together. n-bit string $\overline{x}$ can be concatenated with m-bit string $\overline{y}$ to create $(n+m)$-bit string $\overline{xy}$. Strings are joined left-to-right.
Example:

$$\overline{x} = 000$$
$$\overline{y} = 11$$
$$\overline{xy} = 00011$$

> **Note**
>
> Concatenation can also be written like $\overline{x}||\overline{y}$ or $\overline{x} \cdot \overline{y}$

2.2 Representing Text

2.2.1 Encoding

Bits can be used to represent other forms of data, such as text. For bits to represent characters, an encoding is needed. The criteria for such is as follows:

- A set of characters to represent
- Length n for bit strings
- Mapping from characters to $\{0,1\}^n$
  - The mapping needs exactly one bit string for each character, and no two characters sharing the bit string

2.2.2 Lexicographic Ordering

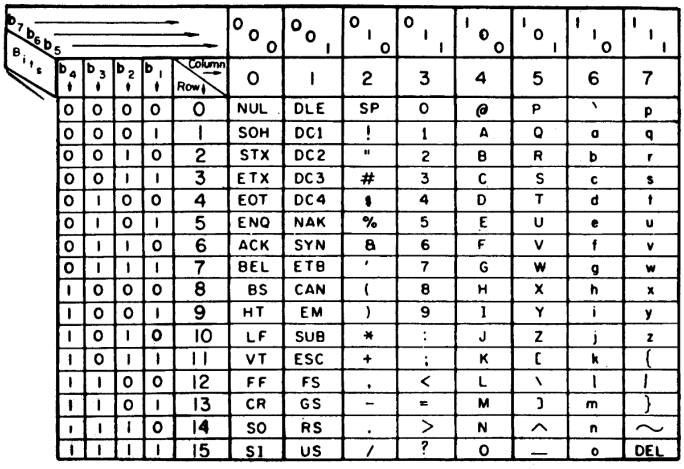
Lexicographic ordering is a common way to order bit strings and follows these rules:

- 0 comes before 1
- Compare strings one bit at a time, from left-to-right
- Where strings differ, the string with a 0 goes first
- If a string is longer, pad the shorter string to the right with empty spaces
- Empty spaces come before 0

2.2.3 ASCII

ASCII is an old, but still used encoding for English text. Some characteristics include:

- 7 bit strings
- 128 characters
- Upper and lower case Latin characters, numbers, punctuation, symbols, space, newline
- Special characters relating to teleprinters like BEL, ESC, and NUL



b_4	b_3	b_2	b_1	Row	000	001	010	011	100	101	110	111
				Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figure 1: USASCII Code Chart

2.2.4 Unicode

Unicode is a modern encoding for many languages and writing systems. Some characteristics include:

- Around 137,000 characters supported
- Support for most writing systems
- Mathematical symbols, punctuation, emoji
- Multiple size encodings

Unicode assigns a hexadecimal string (code point) for each character. It also has several different encodings using different-sized bit strings.

- UTF32, using 32 bits for each character
- UTF16, using one or two 16-bit strings. This makes it *variable length encoding*
- UTF8, using between one and four 8-bit strings. It is also *variable length encoding* and is backwards compatible with ASCII. UTF8 is the most common encoding

2.2.5 Interpreting Text

Characters are often needed in sequence, forming words. These are called character strings and are one long string of bits. Interpreting these bits requires knowing:

- How long each character representation is
- When to stop

For fixed-length encodings, knowing when to stop is sufficient. Variable-length encodings require a way to recognise when a character representation is finished. Most modern programming languages store the amount of characters alongside the string. C however, uses *null-terminated strings* which store a reserved bit string 00000000 (NUL in ASCII) at the end to indicate the string has ended.

3 Week 3: Propositional Logic

3.1 Recursion

3.1.1 Recursive Definitions

An object described in terms of itself is called *recursive definition*. For example, the factorial of $\mathbb{N}$ can be defined by:

$$n! = \prod_{j=1}^{n} j = 1 \cdot 2 \ldots (n-1) \cdot n$$

$n!$ can also be defined recursively by:

$$n! = \begin{cases} 1 & : n = 1 \\ n(n-1)! & : n > 1 \end{cases}$$

Another example of recursion is the Fibonacci Sequence:

$$f(n) = \begin{cases} 1 & : n = 1 \\ 1 & : n = 2 \\ f(n-1) + f(n-2) & : n > 2 \end{cases}$$

3.1.2 Recursive Cases

The two main parts of a recursive definition are:

- **Base cases**: evaluated without any reference to object
- **Recursive cases**: Cases that refer back to object definition

At least one base case is required, but there may be several. Recursive cases must always resolve back to base cases to be well-defined.

3.2 Propositional Logic

3.2.1 Propositional Logic

Propositional logic involves:

- **Propositions**: Statements which are *true* or *false*
- **Logical connectives**: operators used to build larger propositions from smaller ones

3.2.2 Propositions

A proposition is a statement that is either *true* or *false*.

- **I like tomatoes** could be true or false
- **All humans are mortal** is true

If a truth value cannot be assigned, it is not a proposition.

- **This sentence is false** is neither true or false, therefore not a proposition

> Note
>
> Propositional logic is not concerned with the content of the statements themselves, only their truth values. Therefore, the symbols p, q etc. are often used to stand in for propositions.

3.2.3 Atomic and Compound Propositions

Atomic propositions are propositions that cannot be broken down further. *Compound propositions* are composed of two or more atomic propositions.

- **It is raining** is an atomic proposition. It cannot be broken down further
- **It is raining and it is cloudy** is a compound proposition comprising of the two atomic propositions:
  - **It is raining**
  - **It is cloudy**

Extra examples of compound propositions:

- **It is raining or snowing** is composed of:
  - **It is raining**
  - **It is snowing**
- **If I hit my head then it will hurt** is composed of:
  - **I hit my head**
  - **My head will hurt**

3.2.4 Logical Operators

Atomic propositions can be combined into compound propositions using *logical operators*. Propositional logic uses the same operators as bitwise operations, namely:

- **NOT** symbolised by $\neg$ (See 2.1.5)
- **AND** symbolised by $\wedge$ (See 2.1.6)
- **OR** symbolised by $\vee$ (See 2.1.7)
- **XOR** symbolised by $\oplus$ (See 2.1.8)

The logical operators listed above function identically to their bitwise counterparts, however propositional logical also introduces some new operators:

- **IF .. THEN** symbolised by $\rightarrow$
- **IF AND ONLY IF** symbolised by $\leftrightarrow$

> Note
>
> For the following truth tables, p and q are semantically equivalent to x and y, and T and F are semantically equivalent to 1 and 0 respectively.

3.2.5 IF .. THEN

IF .. THEN is a binary operator that results in *true* if q is *true* when p is also *true*. When p is *false*, the result is always *true*.

Table 5: IF .. THEN Truth Table

p	q	$p \rightarrow q$
F	F	T
F	T	T
T	F	F
T	T	T

3.2.6 IF AND ONLY IF

IF AND ONLY IF is a binary operator that results in *true* if **both** operands are the same value.

Table 6: IF AND ONLY IF Truth Table

p	q	$p \leftrightarrow q$
F	F	T
F	T	F
T	F	F
T	T	T

Note

The IF AND ONLY IF operator is an inverse of the XOR operator.