Computer Science Notes

Alexander Neville

March 25, 2022

Contents

1 Fundamentals of Programming

An algorithm is a series of steps underlying every process including programs. An abstract view of any program/algorithm is *input -¿ process -¿ output*.

1.1 Pseudocode

Rather than try and solve a problem while implementing a program, it can help to try and outline the solution in a abstract manner and then write the code to solve the problem. Pseudocode is a flexible method of planning an algorithm that is easier to read and understand, while keeping in the programming mindset. Pseudocode is written like a program, with data types, flow control, operators etc.

1.1.1 Syntax

Comments:

Anything following a `#` symbol is a comment.

Data Types:

Pseudocode is not strongly typed so there is generally no need to specify the data type on initialisation. Pseudocode variables typically fall into these categories:

- *integer* (any unbounded whole number)
- *real/float* (any number with a fractional part)
- *boolean* (a true/false variable)
- *character* (a single digit represented in ASCII or similar)
- *string* (more than one character in a row)

NB. all data-types will have a constructor eg. *int("2")* or *str(2)* for easy conversion. NB. all iterable series in pseudocode are 1 based.

Assignment:

The `=` sign is often used. A left arrow `<-` is sometimes used to show the flow of data. Variable names are all lowercase, while constants are defined with uppercase identifiers.

Operators:

Like other languages, the symbols `*/-+` are used. In addition these rules tend to apply:

- Assignment operators are rarely used as they tend to obfuscate the step by step approach to problem solving; code is usually broken into *input -¿ process -¿ output* cycle.
- To raise a number to a certain power, this syntax is used: `x**n`, where x is the base and n in the exponent.

Keyword Functions:

Besides the symbolic operators there are a number of functions and statements built into pseudocode. Here are a few:

- *div* (floor divide operator, returning a whole number)
- *mod* (find the remainder of a certain div operation)
- *length()* (find the length of a string)
- *string.substring(index, index)* (return the string found between inclusive indices)
- *string.find(substring)* (return the first index of substring if found, else return -1)
- *ord("x")* (find the ASCII value of a character)
- *chr(97)* (find the character associated with a number)

Input and Output:

Input can be taken by assigning the result of `USERINPUT` to an identifier. The `OUTPUT` statement can print data. `INPUT` is sometimes used when the is not necessarily entered by a person.

```
sensor_data <- INPUT
name <- USERINPUT
OUTPUT name, sensor_data
```

1.1.2 Flow Control

Pseudocode can combine its syntax with flow control mechanisms to build more functional programs.

Relational Operators:

Pseudocode uses all the typical relational operators, eg. ¡=, ¿=, ¡, ¿. In addition a single = sign is used as a comparison operator as `<-` is typically used for assignment. The `<>` symbol is used as not equal to operator.

Boolean Logic:

`AND`, `OR` and `NOT` can be used to create logical statements.

Conditional Statements:

This is how a typical condition is constructed:

```
IF (condition) THEN
    ....
ELSE IF (condition) THEN
    ....
ENDIF
```

Switch Statements:

If multiple options are dependent on the value of a single variable, a switch statement can be used.

```
CASE variable of
    value:
        ....
    value:
        ....
ENDCASE
```

Iteration:

Predicated *while* loops are constructed like this:

```
WHILE condition
    ....
ENDWHILE
```

Do while loops are also a possibility, useful if code must run at least once.

```
REPEAT
    ....
UNTIL condition
```

If the number of iterations is know, a *for* loop can be used

```
FOR variable <- value TO value STEP value
    ....
ENDFOR
```

1.1.3 Data structures

Arrays are a collection type which are usually filled with data of one type and unite values under one identifier. Individual values can be accessed with indexing eg. `array[x]` will return the value at position *x* along *array*. Arrays may be multi-dimensional and values can be addressed like this: `array[x][y]`.

1.1.4 Subroutines

A subroutine is a named block of code within a program. A subroutine may perform an operation as in a *procedure* or return a value to the main program as a *function.* In pseudocode a subroutine can be defined and called like this:

```
SUB procedure
    ....
END SUB

SUB function(input)
    ....
    RETURN output
ENDSUB

procedure
data <- function(data)
```

1.1.5 Files

To store data permanently, It must be written to a file on disk. A file contains *records* (rows) with many *fields.* Data can be read from a file like this:

```
OPEN file for reading
FOR line <- 1 TO num_lines
    record = READLINE(file, line)
    OUTPUT record[1], record[2], ...
ENDFOR
CLOSE file
```

1.1.6 Exceptions

To define behaviour for an error condition, exception handling can be used.

```
TRY
    ....
EXCEPT
    OUTPUT "there was an error"
ELSE
    OUTPUT "end of statement"
ENDEXCEPT
```

2 Problem Solving and Theory of Computation

2.1 Problem Solving

Besides writing larger programs, computing has many applications involving smaller problems. A *puzzle* is a problem that is solved by selecting the right inputs. This process can be performed computationally. The problem may be *specific*, having a certain number of inputs (eg. 3), or *general*, having n number of inputs. These values can be used to calculate the efficiency of an algorithm.

2.2 Strategies

There are some common strategies for solving logic/computational problems:

- *exhaustive*, can be described as systematic, is a *brute-force* technique. The inputs are not selected intelligently, based on higher probability of solving the problem, but rather randomly or in some arbitrary order.
- *divide-and-conquer*, works best with partially solved puzzles, eg. sorted list. The number of inputs is repeatedly split and the more probable path is taken.

2.3 Structured Programming

In order to ease development and make maintainable programs, an algorithm is divided into smaller parts.

2.3.1 Block Structure

In block-structured languages, an algorithm can be broken down into the repeated use of just three structures:

- *sequence* - a block of code composed of one instruction after the other (single thread of execution)
- *selection* - the use of a conditional statement to execute certain sequences depending on an event
- *iteration* - the use of abstract *jumps* to repeat a sequence of code

Modern programming languages use syntax elements to make these *blocks* apparent. Curly brackets, `{}`, or indentation and significant white space might be used to make code blocks visually distinct.

2.3.2 Modularisation

An algorithm is repeatedly broken down into smaller parts until each can easily be implemented in a single *sub-routine*, sometimes called a *module*. This is called *top-down* design. The advantages of this technique include:

- individual module/unit testing
- reusable and distributable modules
- many people can work on a project simultaneously

2.3.3 Hierarchy Charts

A hierarchy chart is a way of visualising how an algorithm is broken down. Each step may be a logical block or a sub-routine that has been programmed. A hierarchy chart does not describe the implementation of a problem, nor the control flow within each module.

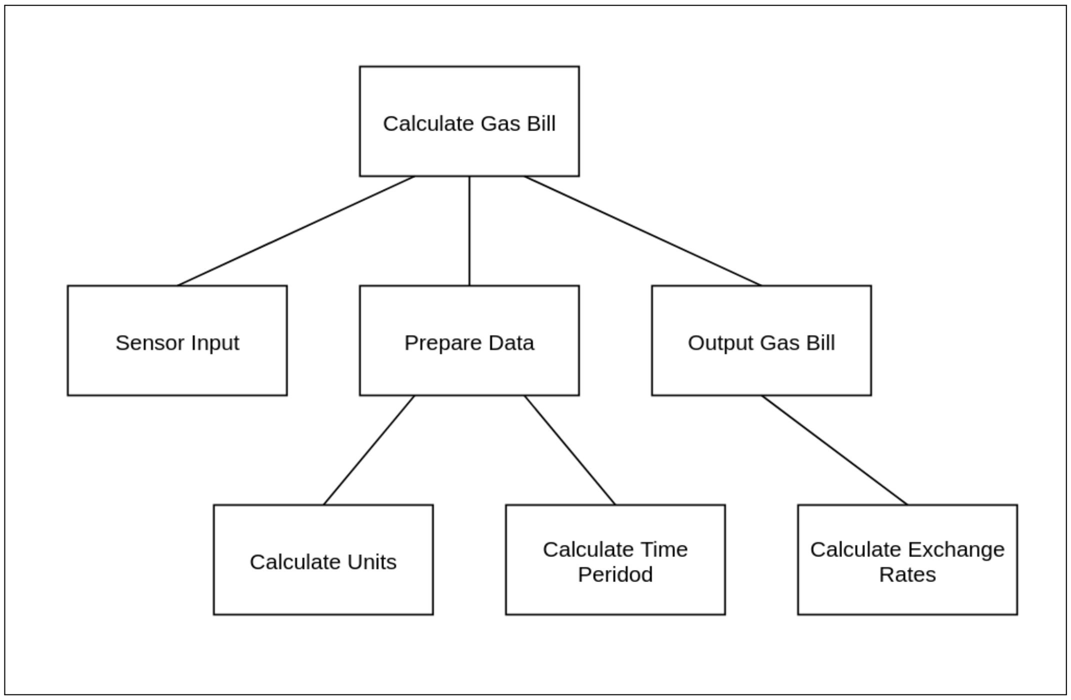

Figure 1: Example hierarchy chart

2.4 Testing

All algorithms should be thoroughly tested to detect problems that could occur under certain conditions. Any inputs should be tested with *normal*, *boundary* and *erroneous* data. Before running a program, it may be *dry-run*, using a trace table.

2.5 Abstraction

Abstraction is the process of simplifying something by removing unnecessary details. This is a common technique in programming, as most high-level operations are made irrespective of the hardware and machine operations that need to take place.

Abstraction by *generalisation* is a technique used to remove context from a problem and equate it to existing problems and scenarios. Therefore, the problem can be worked on in a theoretical manner and once solved, applied to the initial problem. Similar is the idea of *problem abstraction*, where the problem is abstracted and generalised to a point where it matches an existing problem and solution.

Procedural abstraction is often used in computing. Once a problem has been solved and implemented, there is no need for that module to be re-written. This is *information hiding*, as the program calling a module does not need to know its implementation. This kind of abstraction depends on the existing implementation of a problem's solution.

Data can also be subject to abstraction. The behaviour of numbers, when subject to mathematical operations, depends on the number's type, eg. float or integer, rather than the program code.

2.6 Composition

Breaking an algorithm down, via any method, is called *decomposition*. The process of combining existing smaller modules to solve a larger problem is called *composition*.

2.7 Automation

not implemented

2.8 Finite State Machines

A *finite state machine* is an abstract view of some computation. Using *states* and *transitions*, an FSM demonstrates how a system responds to an event under various conditions (states).

A state is represented with a circle. States are joined by an arrow (direction is important), representing a transition. A transition is usually labelled with a transition *condition*.

The start state is marked by a short arrow, with no connection to another state. The end or *acceptance* state is a double circle. See the diagram below.

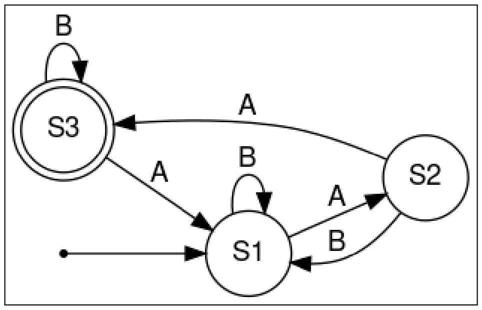

Figure 2: FSM with three states

The typical FSM diagram can also be represented by a *state transition table*, which lists all of the possible transitions. The table for the diagram above would look like this:

current state	input = A	input = B
S1	S2	S1
S2	S3	S1
S3	S1	S3

3 Data Representation

3.1 Number Systems

3.1.1 Sets of Numbers

- Whole Numbers = Z (negative or positive integers)
- Natural Numbers = N (integers above 0)
- Rational Numbers = Q (can be expressed precisely as a fraction)
- Real Numbers = R (anything that can be expressed numerically, includes irrational numbers)
- Ordinal Numbers: *first*, *second*, *third*

3.1.2 Decimal

Decimal (Base 10) is the number system we use on a daily basis. It may originate from the ten fingers and toes humans have. With current technology, it is impossible to use base 10 in computer systems.

3.1.3 Binary

Binary is the number system used in computing and understood by a computer's processor. It is used because of the relative ease of distinguishing between just two states: *on* & *off*. The disadvantage of this system is representing large amounts of complex data. With only two states, many binary bits in sequence are needed to represent real world data. In decimal, 10 values can be represented with one character and the total number of values available with n characters is:

- 10^n

While, with binary, the number of available values when using n bits is:

- 2^n

So in order to get an equivalent range of values, the value n must be larger when using binary. This is manageable in a computer and the advantages significantly outweigh the disadvantages, however this is difficult for people to work with and understand. Binary is long and repetitive, making working with binary slow and error prone.

3.1.4 Hexadecimal and Octal

Hexadecimal (Base 16) and Octal (Base 8) are used to make working with computers easier. The range of values that can be represented with n place values is:

- 16^n for hexadecimal
- 8^n for octal

These number systems in particular are used to represent binary values more concisely, while being easier to convert to and from binary than decimal numbers. Both 8 and 16 are powers of 2; this means that numerous individual bits of a binary number are represented by a single character or value in one of these number systems.

- 1 octal place value can represent exactly 3 binary bits
- 1 hexadecimal place value can represent exactly 4 binary bits

The same is true of Base 4 and Base 32, but these are not used nearly as frequently.

Uses of Hexadecimal or Octal number systems:

- Colour Codes
- MAC Addresses
- IPv6 Addresses
- Assembly Language
- Unix File Permissions

3.2 Two's Complement

There are a number of different techniques for handling negative numbers in computer systems. *Two's Complement* is a common method of doing so, as two's complement numbers can be treated like a regular value during computation.

In Two's complement binary, the most significant bit of a number is treated as negative, hence:

- *if a number begins with 1, its value will be negative*
- *if a number begins with 0, its value will be positive*

In the table below, a few examples of a 4-bit two's complement number are given. The whole number is written in the left column and is also broken down into place value columns to the right.

number	-8	4	2	1	decimal value
1000	1	0	0	0	-8
1111	1	1	1	1	-1
0000	0	0	0	0	0
0101	0	1	0	1	5
1011	1	0	1	1	-5

The decimal value in the right-hand column is the sum of all the place value columns containing a 1. This procedure is broken down below.

```
1111 = (-8) + 4 + 2 + 1 = -1
1000 = (-8) + 0 + 0 + 0 = -8
0000 = (-0) + 0 + 0 + 0 = 0
0101 = (-0) + 4 + 0 + 1 = 5
1011 = (-8) + 0 + 2 + 1 = -5
```

With n bits, the range of values you can represent is:

$$2^{(n-1)} \text{ ... } 2^{(n-1)} - 1$$

To obtain the two's complement of a number (negative to positive or vice versa), flip all of the bits and one.

The benefit of the two's complement system is that it maximises the range of values which can be represented by a *word* of a certain length, eg. using a designated sign bit a value for zero and negative zero must be stored, which is not needed and can complicate some calculations.

Computers generally rely on two's complement to perform subtraction, using only the addition circuits at their disposal. In order to subtract one value from another:

- the number that must be subtracted is converted to its two's complement
- the two numbers are now added to one another
- therefore $37 - 9$ becomes $37 + (-9)$

3.3 Fixed Point Binary Numbers

In a fixed point binary value, some bits fall before and after the point. The position of the point is usually determined as needed. Using such a system, any bits before the point are treated as usual. For any bit after the point, its value is 2^{-n}, where n is the position from the decimal point.

Here is a demonstration of this principle:

n	power of 2	value
1	2^{-1}	0.5
2	2^{-2}	0.25
3	2^{-3}	0.125

The position of the decimal point within a fixed point binary value can determine the properties of the number: *range* vs *precision*

3.4 Floating Point Binary Numbers

Fixed point binary numbers only offer limited precision, unless an extra-ordinary number of bits are used. Many bits are needed to represent very small fractions and many bits are needed to represent very large numbers, even if fixed point binary is not applied. *Floating Point* binary values work like scientific notation, making them suitable for extremely large or small numbers. In such a number the bit pattern used is split into two parts: the *mantissa* and the *exponent*.

3.4.1 Conversion

This table shows how to convert `01101 011` into a fixed point binary number. When given a floating point number, the point's default position is just after the first bit (as in scientific notation). The mantissa records how many places to the right the point needs to move. (Nb. on the first row, the default position of the point is shown)

mantissa	exponent	exponent decimal
0.1101	011	3
01.101	010	2
011.01	001	1
0110.1	000	0

3.5 Character Encoding

Human readable characters need to be represented numerically for use in computer systems. The given numerical value for a character can be expressed in decimal, but binary is always used by computers. Two very common standards for character encoding are *ASCII* and *Unicode*. Note that not all data is encoded with these standards; compiled code and many image formats are *raw binary* data. This kind of data cannot be read by a human or displayed in a text editor.

3.5.1 ASCII

The first major encoding standard was ASCII. ASCII was designed to be a 7 bit standard, allowing 128 different characters to be represented, while leaving space for a parity bit within each byte. Later on, an eighth bit was added to extend the number of possible characters which could be used. The new 8 bit ASCII maintained compatibility with the original standard; the first 128 characters are the same. 8 bit **ASCII** is sometimes called **UTF-8**.

3.5.2 Unicode

As the internet became pervasive and computers in all parts of the world became connected, a new standard was needed to manage more languages and their character sets. *Unicode* was developed to solve this problem. It was initially a 16 bit standard, allowing 65,536 different characters to be represented, enough for multiple character sets. The first 8 bits of this character set matched those of 8 bit **ASCII**, so there is some compatibility. **UTF-32** now exists, offering over a million different individual characters. The downside of these enlarged standards is the size. **UTF-32** is twice as large as **UTF-16** and twice as large again as ASCII. This means text encoded with Unicode will take up more storage on a computer and take longer to transmit.

3.6 Error Checking and Correction

Errors can occur when data is read, inputted or transmitted. There are many ways to identify when an error has occurred and how errors can be corrected.

3.6.1 Parity Checking

Parity bits are a common method for protecting against errors during communication. 7 bit *ASCII* lends itself towards transmitting a parity bit within each byte. Even or odd parity may be used. The parity bit ensures that the total of all the bits (including the parity bit) is an odd or even number in accordance with the technique being used. Upon receiving data, the recipient can check the total of the bits. If the total does not correspond to the chosen parity, an error has occurred. This method cannot detect *transposition errors* (when the order of bits is changed), nor are they able to identify errors affecting more than one bit.

3.6.2 Majority Voting

When majority voting is used, each bit is transmitted repeatedly. An odd number of repetitions must be used so there is not a *tie*. The received values for each bit are compared and the *popular* result is taken to be the true value of that bit. Hence, there is a means of correcting potential errors. Eg. one of three transmissions of a single bit may differ from the two others. It can be concluded that this bit is erroneous and its value discarded. Transmitting data repeatedly, for the purpose of error detection and correction, significantly increases the time taken to send a certain communication. Majority voting is often infeasible, because of the time it takes.

3.6.3 Checksum

A checksum is a thorough method of error detection. An algorithm is applied to a piece of data before it is sent, the result being the checksum. The checksum is transmitted with the block of data. Upon receipt of the block, the same algorithm is applied to the data (which might have been corrupted). The checksum calculated by the receiver is compared with the transmitted checksum. If the two values do not match, an error has occurred. If the two values do match, it is likely that no error has occurred.

3.6.4 Check Digits

The role of a *check digit* is to prevent *transcription* errors (or other similar I/O errors) of identifiers and other short pieces of data. Check digits are often attached to barcodes, ISBNs and credit card numbers. A checkdigit is calculated by running a number through an algorithm. The result is usually printed alongside the data, wherever it appears. Devices like barcode readers can calculate the check-digit based on the obtained number and compare it too the existing checkdigit. If the two values do not match, an error has occurred. If the two values do match, it is likely that no error has occurred.

3.7 Analogue and Digital Data

Analogue data is *continuous*, having physical quantities which are changing constantly. This kind of data can only be truly represented by a continuous range of values. For the sake of digital representation, analogue waveforms and other continuous sets of data are sampled at intervals, or in certain places. Each of these samples contains data which is quantifiable, so it may be used and stored by a computer. Therefore any digital representation of analogue data is an approximation of some quantity at a certain time and place.

3.8 Image Representation

Image data can be broadly categorised into two types: *photographs* and *digital graphics*. While there is no scientific definition for either of these terms, a photograph is generally captured by a camera, while computerised digital graphics are typically designed or generated. Photographs are usually stored as *bitmap* images, while *vector* graphic files are more suitable for computerised images.

3.8.1 Bitmap Files

Bitmap images are composed of *pixels*. A 'picture element', or a *pixel*, is the smallest identifiable area of an image and each pixel will contain information about the image at that point.

A raster (bitmap) file itself contains all of the pixel data making up the image and some **Metadata** needed to reconstruct the image. A bitmap file has a *size*, which is a number of pixels expressed in terms of *width * height*.

The *resolution* is the number of *dots per inch* (dpi). An image of greater *size* will have a higher *resolution* when displayed in the same space/scale on a screen.

Bitmap files store an approximation of real world analogue data and they are ideal for photographs, which have constantly changing colour gradients and no distinct boundaries.

3.8.2 Limitations of Bitmap Files

The *size* of an image (in pixels) does not determine the area its is displayed on. Should an image need to be displayed in a physical form that is greater than the original dimensions, the resolution (ppi/dpi) is diminished. For simple graphics, a bitmap file's size may exceed that of a vectorised file format.

3.8.3 Colour Depth

In a bitmap file, each picture element has an associated colour code. The length of this value determines the number of colours which can be represented.

A common colour depth is 3 bytes (24 bits), where each byte corresponds to one *RGB* channel. Each channel has 8 bits and so the number of possible colours (in each channel) is:

$2^8 = 256$

As there are three channels, the total number of colours is:

$256^3 = 16777216$

A colour depth of 3 bytes offers more colours than the human eye can distinguish between, hence there is little benefit using a greater depth. A larger colour depth increases file size, so it is sometimes advantageous to use a reduced set of colours.

3.8.4 Metadata

This is data stored in the header of a bitmap file, containing all the information needed to display the image. Eg. columns, rows, colour depth, etc.

3.8.5 Vector Graphics

A vector file consists of a drawing list, containing a list of all the shapes that need to be drawn to *construct* the image. Unlike a bitmap, the shapes listed in a vector file can be redrawn and the image constructed proportionally to suit any display size. This makes vector graphics ideal for images which may need to be displayed in many places at different sizes.

The file size of a vector graphic will depend on the number of objects which have to be drawn, rather than the size and quality of the image. Photographs cannot be represented with vector graphic files easily, because of the complex shapes and many colour gradient, with few distinct boundaries.

3.9 Audio Representation

Sound is a type of analogue data, which is - in nature - a continuous wave. This data must undergo analogue to digital conversion.

3.9.1 Sample Rate

In order to represent a continuous sound wave as discrete digital data, many quantised *samples* must be taken at regular intervals. The frequency of the recording, also called the *sample rate*, is the number of samples per second. The greater the sample rate, the closer the digital representation of the audio is to the original sound.

3.9.2 Sample Depth

As well as increasing the number of samples stored, audio quality can be improved by increasing the *bit depth*. The bit depth is the number of bits used to store the amplitude of the sound at a given sample. The higher the bit depth, the closer the amplitude to its original value. *Nb. amplitude is often represented on the Y axis, against time on the X axis*

3.9.3 Nyquist's Theorem

Discrete digital data cannot perfectly represent all of the properties of a continuous analogue waveform. In 1928, Harry Nyquist theorised that a recording must be sampled at twice the maximum frequency of the analogue sound to produce an accurate recording. For the human ear, the maximum audible frequency is 20,000Hz. Therefore, audio is often sampled at 44,100Hz, beyond which there is no apparent difference in sound quality to humans.

3.9.4 Audio Storage and Files

Audio which is recorded from a live source is stored and can be played back using a DAC and a speaker. There are many common file formats for sound and video, many of which use compression. Sound files may also contain metadata about the file.

Sound may also be 'stored' as a MIDI file, a set of steps which can be interpreted by software to synthesise new sound. It is primarily a tool for music artists to create new music which can later be recorded. As MIDI files do not try to replicate analogue data with lots of samples, they may be smaller in size to a recording of similar length and quality.

3.9.5 Recording

1. A computer peripheral (microphone) is used to convert a sound wave into an oscillating electrical signal.
2. The electrical signal is suitable for analogue to digital conversion.
3. An ADC will *sample* this signal to a given frequency. Any data between samples is lost.
4. The ADC will approximate the amplitude of the sound for each sample.
5. The output - discrete digital data - can be stored in typical computer storage devices.

3.9.6 Playback

1. In order to present the stored data, it must be converted back to an analogue form. A *DAC* (digital to analogue converter) can be used to this effect.
2. The recorded amplitudes for all the samples are converted into an electrical signal (a voltage) at the same frequency as the sample rate.
3. The voltage changes are converted into a sound wave by a speaker.

3.10 Compression

Image and sound files can be very large and repetitive. Generally, a small reduction in quality is tolerated, making these files good candidates for lossy compression, where the quality is somewhat reduced to achieve greater compression ratios.

Text files, including programs, are rarely as large as other types of data, however reducing their file size is sometimes needed. It is essential that compressed text files can be recreated without any loss in quality. Lossless compression is a compression method which maintains the exact quality of the uncompressed data, so it can be read exactly as intended after compression and subsequent uncompression. The compression ratios of Lossless compression techniques rarely equal those of lossy compression.

3.10.1 Lossless

Lossless compression ensures the original file can be recreated from the compressed file. Therefore the compressed file must convey *exactly* the same information as the original in a slightly different way. Lossless compression algorithms typically reduce repetition within files as a means of compression.

Most Lossless compression methods are only effective when there is significant repetition. In some cases, when there is not sufficient repetition, negative compression can occur and the size of the compressed file exceeds that of the original file.

Run Length Encoding:

Using RLE, file size is reduce by removing runs of identical data. Each run of data is replaced with the original piece data and the length of the run (number of repetitions).

Data which does not have long runs of identical data is not suitable for this type of compression. RLE can be very useful in the compression of sound files, as a single sound played for even a short amount of time may result in many identical samples.

Dictionary Compression:

Dictionary compression is a more flexible type of lossless compression. The compression algorithm creates a *dictionary* associating frequently repeated pieces of data to an index. Any occurrence of an indexed piece of data are replaced with the corresponding dictionary index.

Nb. The compressed file must include the dictionary that is used. Dictionary compression works best with larger files, where the size of the dictionary is offset by the amount of repetition removed.

3.10.2 Lossy

Unlike lossless compression, lossy compression is irreversible, as data is **permanently** removed from the file. This means that the effectiveness of lossy compression is not dependent on an amount of repetition within the file. However, this method reduces the quality of the file which is compressed, making it unsuitable for text and similar sorts of data.

3.11 Encryption

Encryption is the process of changing data so that it is only readable to the intended recipient. A cipher is an algorithm which encrypts some data. Decryption of cipher text requires the *key* used to encrypt the file and the encryption method must be known. The original data is referred to as plain text and the encrypted data is referred to as cipher text.

3.11.1 Caesar Ciphers

This is a very old, basic cipher, using character replacement. One character in the cipher text always represents the same plain text character.

A shift can be used to quickly generate a cipher. The ciphertext value of a character is found by moving a certain number of places through the alphabet. In this case, the key is the shift required to move from the plain-text to the ciphertext. This process can be reversed by the recipient to decrypt the data.

A substitution cipher may also be used. With such a cipher, the letters are randomly replaced. This introduces more complexity as there is not a single key that applies to the whole data. To decrypt such a cipher the 'key' for each character must be known to the recipient.

All Caesar ciphers are fairly easy to decipher without the key. In the case of a shift cipher, brute force is possible as there are only 25 possible keys. Frequency of the ciphertext characters can be analysed as every occurrence will refer to a certain plain text character. In English, certain characters and combinations are more frequent so the most common cipher text characters can be found and decrypted.

3.11.2 Vernam Ciphers

The Vernam cipher is a more secure algorithm. It requires a key in the form of a one-time pad. *One-time* means it should only be used once, to ensure its randomness. The key must be as long, or greater than, the plain text to be encrypted.

Encryption:

- Plain text is aligned with the beginning of the one-time pad

- The characters are represented numerically, in binary
- A logical XOR process is performed on the plain text and one-time pad bit patterns
- The resulting bit pattern is translated back into a character

Decryption:

- To decrypt the cipher text, the same one-time pad must be used
- The two strings are aligned
- Both strings are converted to binary
- The logical XOR operation is carried out
- The resulting bit pattern is converted back to a character, which should equal the initial plain text character

The one-time pad which is used must be generated randomly, only this can guarantee the randomness of the ciphertext. If the pad is new and random, the ciphertext is completely unbreakable at the point it is encrypted. This cipher is mathematically unbreakable, if all standards are upheld. To make this cipher more efficient, an amount of pad may be exchanged between parties before any messages are sent. Each message sent will use the next unused section of pad as the key.

Vernam ciphers are not always used in computer systems, even though it is mathematically secure. Given current computing power, many other algorithms - which can be cracked - will take long enough that any efforts to crack the encryption will not be viable. This is called computational security.

4 Hardware and Software

Hardware is the name given to physical equipment that constitutes a computer. Meanwhile, software is program code stored digitally and used to perform a task.

4.1 Classification of Software

4.1.1 System Software

System software is the collection of software needed to run a computer system, including an operating system. A general purpose operating system will usually ship with additional software to make the computer more functional. There are other types of system software.

1. Operating Systems

   An operating system *(OS)* is a piece of software designed to manage a computer's hardware on behalf of running applications and ultimately the computer user. The operating system interacts directly with the system's resources, exposing available operations to other applications, a good example of a programming *API*. The benefit of an operating system is versatility. With an *OS* installed there is no need for each application to perform low level operations and the computer will be able to perform many different tasks without reprogramming.

   Common OS tasks:

   - multi-processing and multitasking
   - backing store management
   - device and driver management
   - hardware diagnostics
   - detecting peripherals
   - interrupt handling

2. Utility Software

   Utility software is often included with an operating system to help maintain the machine, but the user may choose to install more of these utilities. Typically, the operating system itself will use the installed utility software, although the user may choose to use the software manually.

   Examples include:

   - disk management
   - virus checking
   - file management
   - compression tools
   - software installer/uninstaller

3. Code Libraries

   Many programming languages have a library of code that is *built-in*, to simplify common operations. This code can be packaged and implemented in many different ways, but the core functions are usually included with the language and stored on a users computer. Some common library extensions are: **.dll** and **.so**. Should a program need functionality from the libraries, it is *imported* in the source code.

4. Translators

   Translators are required to transform written source code into executable machine code, with various steps along the way. The different types of translators are covered below.

4.1.2 Application Software

Application software is designed to work on a certain operating system and perform tasks for the user. This type of software may be shipped along with an operating system, although a user may choose to install or remove this type of software on a regular basis. Application software is developed in different ways for different customers. Here are some common ways software is made and distributed: *General Purpose*, *Specific Purpose* and *Bespoke*.

1. General Purpose
   - Includes mainstream products like office suites or graphics programs
   - A user will install this kind of software to help with whatever specific purpose they have in mind
   - This type of software will be used by many people to achieve different results
2. Special Purpose
   - This is also a type of application software, but unlike general purpose software, this type of software serves a more specific audience
   - This type of software is still widely available and its specificity does not reduce its pervasiveness
   - E.g. a python IDE is specific software, but many people will use this software.
   - This type of software does not necessarily match the exact requirements of a person or business, as this type of software is published without a specific customer in mind and it may be disseminated to many people
3. Bespoke Software

   A business may choose to have software developed specially for them by some software developers. This is called *bespoke software*.
   - This kind of software is generally special purpose, as there is no need to create new general purpose software
   - Bespoke software means it not only serves a specific purpose, it is also created directly for a specific customer
   - The benefits of a bespoke system is ease of use and specificity (no redundant/missing features, because the client told the developer what to include)
   - The downside of this type of software is the time and cost associated with developing the software after the need is identified
   - Additionally, the software may not be stable if it has not been tested extensively by others

4.2 Programming Language Specification

4.2.1 Low-Level Languages

The two main types of low-level language are *machine code* and *assembly language*.

1. Machine Code

   Machine code is written entirely in binary and as a result it can be executed by a computer without the need for translation. At this level, abstraction is limited and a program consists of the set of operations the processor must perform.

Machine code instructions are composed of an *opcode* and associated *operands*. The available opcodes depend on the processor's *instruction set* and so machine code is specific to the platform it was designed for, therefore machine code is not *portable*.

Binary is repetitive and lengthy, making mistakes like transcription errors very common. Once a program has been developed, the machine code is difficult to understand and debug.

2. Assembly language

   Assembly language was designed to make working with native machine code easier. Opcodes are replaced by mnemonics, while hexadecimal and decimal numbers are often used in place of operands. Assembly language reduces the amount of errors made when inputting code and makes the code easier to read and understand.

   Generally speaking, each line of code in an assembly language program corresponds to a single machine code instruction; the extent of assembly language abstraction is limited. This means that assembly language is processor specific, like machine code. In addition, assembly code shares the same detailed approach to programming, as each line of code is a single processor instruction, rather than part of an algorithm.

4.2.2 High-Level Languages

As *operating systems* developed, it became possible to write programs which could be run on many platforms. The languages used to write these programs became known as *high-level* languages. A high-level language will need to be converted, or *translated*, into executable code for each platform it is used on.

High-level languages are much easier to read, write and understand, thanks to the *abstraction* of factors like computer hardware and system resources. Code written in a high-level language more closely models spoken language and uses common symbols like `+` and `-`, along with named variables, comments and indentation to improve ease of use.

High-level languages are much more suitable for designing complex algorithms, as each line describes a single step in the problem, rather than the processor operations needed to make it happen.

A programming *paradigm* is a way of classifying high-level programming languages. The *Imperative* programming language paradigm includes both the *procedural* and *object orientated* techniques. Meanwhile, the *declarative* paradigm includes languages like SQL and the *functional* programming style.

4.2.3 Abstraction

High-level languages are more *abstract* than low-level languages. Each line of python contains far more operations than a line of x86 assembly. Should a process need to be made very efficient, the very minimum level of abstraction should be used. If some process involves specific hardware, it may be useful to use a low-level language to get the best performance from it.

4.3 Programming Language Translation

Besides native machine code, all other types of programming language, including assembly, need to be converted into a format the computer can understand. This process is called *translation*. The two most common methods of translating source code written in a high-level language into object code are *compilation* and *interpretation*.

4.3.1 Assemblers

While assembly is a type of low-level language, it requires translation. The tool used to do this is called an *assembler*. The result of running assembly code through an assembler is object code. Object code may need to be run through a *linker* program to make executable code (not on specification).

4.3.2 Compilers

A compiler is a program that can translate code written in a high-level language into executable code. The compiler program runs over inputted source code, performing a series of checks and identifying how to construct the object code output.

A program written in a compiled language is usually distributed by sharing the compiled object code, which can be run in the absence of the compiler. This means the original code can be kept private or *closed source*.

Advantages of a Compiler:

- Errors are detected before translation
- Compiled code can be run without the need for translation, making execution faster
- Compiled code can be distributed without compromising the source code

4.3.3 Interpreters

An interpreter translates a program into executable machine code instructions at run-time. Sometimes there is an intermediate type of compiled code, called *bytecode*.

A typical interpreter will scan source code in advance for syntax errors, and subsequently translation happens line-by-line. The interpreter will call subroutines within its own source code to handle high-level instructions in the input code.

As the interpreter works line by line, some code may be translated and run before an error is reached.

Advantages of an Interpreter:

- No long periods of compilation
- Identifying errors and debugging the program is easier

4.3.4 Bytecode

In order to improve the performance and portability of high-level code, *bytecode compilation* is sometimes used (usually in interpreted languages). This divides the compilation process into two parts:

1. *bytecode compilation*
2. *machine code translation*

Implementations of this process vary. Some languages like **java** have a portable *virtual machine*, capable of interpreting bytecode. Once source code has been compiled into bytecode for the java virtual machine, it can be run anywhere the *JVM* is installed.

In the case of java, bytecode for the *JVM* is distributable. With the standard python implementation, the bytecode compiler and the interpreter are not separable; bytecode compilation and translation happen at run time. As a result the source code must be shared to distribute the program. It is possible to present python bytecode in human readable form. Here is a program that demonstrates this:

```
import dis

def hello():
    print("hello")

hello()
dis.dis(hello)
```

The output of the script is as follows:

```
hello
  7           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('hello')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

4.4 Boolean Logic & Algebra

4.4.1 De Morgan's Laws

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

4.4.2 General Rules

$$X \cdot 0 = 0$$

$$X \cdot 1 = X$$

$$X \cdot X = X$$

$$X \cdot \overline{X} = 0$$

$$X + 0 = X$$

$$X + 1 = 1$$

$$X + X = X$$

$$X + \overline{X} = 1$$

$$\overline{\overline{X}} = X$$

4.4.3 Commutative Rules

$$X \cdot Y = Y \cdot X$$

$$X + Y = Y + X$$

4.4.4 Associative Rules

$$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$$

4.4.5 Distributive Rules

$$X \cdot (Y + Z) = X \cdot Y + X \cdot Z$$

$$(X + Y) \cdot (Z + W) = X \cdot W + X \cdot Z + Y \cdot Z + Y \cdot Z$$

4.4.6 Absorption Rules

$$X + (X \cdot Y) = X$$

$$X \cdot (X + Y) = X$$

5 Computer Organisation and Architecture

5.1 Harvard & Von Neuman Architecture

Many early computers could only perform one operation. The *Stored Program Concept* allows a computer to store its instructions in re-programmable memory. This means that a computer could be instructed to perform a new operation without the disassembly of the machine. There are two main types of computer architecture designed with this general purpose computing paradigm in mind.

The *Harvard* architecture separates instructions and data, having a dedicated area of memory for each. The advantage of this setup is the computer's ability to fetch data and instructions simultaneously. In addition, a processor with this architecture can have a custom memory configuration depending on its purpose. This architecture is common among embedded systems.

The *Von Neuman* architecture has only one area of memory for both instructions and data. Instructions and data have to be fetched serially, which is often slower. However, this architecture is much more flexible and is used in most general purpose personal computers and mobile devices.

5.2 Motherboard

A Computer's motherboard connects all the components of a computer together. The Central Processing Unit (*CPU)* is housed on the motherboard. The motherboard also houses the computer's interfaces to external components, including the *RAM* and *IO* devices. Processor buses join components on the motherboard, allowing the *CPU* to control the operation of the whole computer.

5.3 CPU Disambiguation

CPUs are implemented on a Integrated Circuit *IC* metal-oxide-semiconductor *MOS* microprocessor chip. Many chips have more than one processor core, these chips are called *multi-core processors.* Each core may have more than 1 thread, creating *virtual cores.* A chip with multiple cores, virtual or physical, will appear to the operating system as multiple **CPUs**. The A-Level specification describes the operation of a single *CPU.*

In short, a *processor* is mounted on a *motherboard* and may have multiple *CPUs*, whether they be physical cores or virtual threads.

Silicon is a useful material for constructing processors. Silicon has semiconductor properties which mean it can behave like a switch, becoming conductive under certain conditions. The flow of charge is how processing works and so changing the conductivity of the chip is required.

5.4 CPU Components

5.4.1 Processor Clock

The system clock generates an oscillating signal, with a frequency in the billions of Hz range. One clock cycle is the time taken for the clock to return to its default position. A CPU operation begins at the beginning of a clock cycle and cannot be complete until the clock cycle has finished. One CPU operation may span many clock cycles.

5.4.2 Control Unit

All the operations and flow of data around the CPU is governed by the control unit. Once an instruction is received, the control unit will organise its execution, including any required mathematical operations in the *ALU*. The control unit also handles data operations, including accessing main memory and general purpose registers.

5.4.3 Arithmetic and Logic Unit

The *ALU* contains circuits capable of most mathematical operations and services requests from the control unit. The ALU will store some information as flags in the *Status Register*, important when making comparison operations.

5.4.4 General Purpose Registers

Modern computer processors have 16 general purpose registers. These are areas of fast, expensive, on-chip storage. In a 64 bit machine, one of these registers is 64 bits in size, although smaller registers are addressable, taking up the least significant bits of their larger counterparts. The data used in *ALU* operations is read from and stored back into these registers.

5.4.5 Dedicated Registers

There are a number of specialised registers within the control unit, necessary for the operation of the computer. Unlike general purpose registers, they are not used to hold the operands and results of ALU operations.

1. Status Register

   Following an ALU operation, *flags* (individual bits) are set in the status register. These Indicate the result of the last operation. Here are some common status register flags:

symbol	purpose
CF	carry bit
PF	parity bit
ZF	zero
SF	sign bit
OF	overflow
AF	adjust
IF	interrupt enabled

2. Memory Buffer Register

   Data that has been copied from main memory is stored here while it awaits an operation. Newly fetched instructions will he copied here initially before they are copied into the *CIR* and decoded.

3. Current Instruction Register

   After an instruction is put in the *MBR*, it is copied into the *CIR* and decoded ahead of execution.

4. Program Counter

   The program counter is responsible for managing the flow of execution. During a fetch, the value of the program counter is copied to the *MAR* so that the next instruction can be retrieved. During instruction execution, the program counter is able to record the address of the next instruction so that program flow can resume after data is fetched form memory, altering the value in the *MAR*. In normal operation the *PC* is incremented so that the next instruction is fetched from the next address in memory, although the contents of the *PC* can be modified to perform a jump and resume operation from elsewhere in the program.

5. Memory Address Register

   The memory address register will hold an address from which data is to be retrieved or written to. Instructions are also fetched from the address held by this register during the *FDE* cycle.

5.5 Motherboard Buses

The *processor* is connected to other motherboard components by *buses*. A bus is a *parallel* wire, through which addresses, data and control signals can flow. Motherboard communication is *synchronous* and the distances involved are short. There are three main buses on any motherboard, which are collectively referred to as the *system bus*.

5.5.1 Address Bus

The address bus is a uni-directional wire, capable of transferring an address from the *MAR* to main memory. The address bus is also used to identify *IO* devices during data input and output.

5.5.2 Data Bus

This bus is bi-directional and allows data to flow between the processor and main memory in either direction. The bus is also used to share data amongst all the other components between these two.

5.5.3 Control Bus

Control signals are sent between motherboard components along this bus. The control bus is bi-directional, meaning these signals can flow either way. Here are some example control signals:

- memory write
- memory read
- interrupt request
- bus request
- bus grant
- clock signals
- reset

Many of these signals may be raised by devices other than the control unit.

5.6 IO Controllers

An *IO* controller sits between the processor and a peripheral device. Device controllers are addressable by the processor and receive instructions and data through the *system bus*. The controller is responsible for converting *CPU* input/output requests to device specific instructions during normal operation.

An *IO* controller is also responsible for detecting and managing connected devices, raising processor *interrupts* when action needs to be taken. This means that the processor can dedicate its time to *FDE* operations rather than device management, significantly improving efficiency.

5.7 Cache

Cache is a small, expensive area of a computer's memory. Cache memory is sometimes located on-chip or at least very nearby. Different *CPUs* will have different cache configurations. In a multi-core processor some of the cache may be shared by all of the processor cores.

A computer will often have different levels of cache, for example:

- L1 cache, with a size between 2 and 64 KB
- L2 cache, with a size between 256 KB and 2 MB

5.8 FDE Cycle

Execution of a machine code instruction can be separated into three distinct phases: *fetch*, *decode* and *execute*. In a modern computer this happens many times per second.

5.8.1 Fetch

The value of the *PC* is copied to the *MAR*. This address is sent to the device's memory along the *address bus*. The value stored in the specified address is returned to the processor on the *data bus*. This value is temporarily stored in the *MBR* while the *PC* is incremented and operation is synchronised with the system clock. The fetch ends as the contents of the *MBR* are copied to the *CIR*.

5.8.2 Decode

The machine code instruction in the *CIR* is interpreted. The instruction is split into opcode and operand sections. Depending on the addressing mode (which is part of the opcode) and the requested operation, any additional data that is required is fetched and stored in one of the general purpose registers.

5.8.3 Execute

The specified operation is performed, having been decoded and identified. If the *ALU* is involved, *status register* flags are set and the results of the operation are stored in either the *accumulator*, general purpose registers or main memory, depending on the device.

5.9 Interrupts

During the *FDE* cycle, the processor will periodically check for *interrupts*. Interrupt signals are carried to the processor via the *control bus*. These signals may originate from *IO* controllers and hence from hardware devices, or they might be raised by running software.

When an interrupt is received and detected, the operating system determines its urgency and how to safely suspend operation and service it. When it is safe to do so, presently executing instructions and their data are pushed on to the *system stack* and processor time is given to the *Interrupt Service Routine*. Once the interrupt is dealt with, control returns to the processor's previous task, provided that the situation was recoverable.

5.10 Processor Performance

A number of factors affect the time a computer will take to perform a given operation or set of operations. No measure alone is able to definitively determine the performance of a computer.

5.10.1 Clock Frequency

The clock speed of a processor governs all *CPU* operations, so theoretically a processor with a higher clock frequency can perform processor actions faster. In practice this is not a reliable single measurement of performance, as other factors dictate the amount of processing that can be performed with a certain number of cycles.

5.10.2 Word Length

The word length of a processor is the number of bits that a *CPU* can operate on in a single action. This value tends to be the same as the size of the processor's registers and the width of the computer's data bus. Modern, general-purpose computers tend to use a 64-bit word length. A processor with a smaller word length will have to make successive fetch operations to work on a similar amount of data to a computer with a larger word length. This will increase the number of clock cycles used to process an amount of data.

5.10.3 Address Size

The size of the address register and the width of the address bus limit the number of addressable memory locations in one operation. With n bits the number of memory locations available is equal to 2^n. With 32 bits this is 4GB of memory. If there is a smaller amount of main memory, the processor will have to copy data back and forth between secondary storage and memory more frequently, to make sure that running processes are performant enough. Multi-tasking may be more difficult with less *RAM*. Under certain conditions this will not affect performance at all. Nb. the length of an address is usually the same as the word length.

5.10.4 Multiple Cores

A processor may have multiple CPUs, whether they be physical *cores* or virtual *threads*. Each CPU can perform a separate *FDE* cycle, significantly improving theoretical processing capability, although not all software will be able to make use of these extra CPUs, limiting the performance gains in most situations.

5.11 Instruction Sets

Computers have different ways of representing available operations to programmers. An instruction set describes the operations a processor can perform with a binary value. An instruction set is specific to a certain processor architecture. The operations that a processor can perform may be similar to those of another processor, although the instruction set, used to trigger those operations, may be entirely different. The instruction set determines how machine code is interpreted and hence written. Each instruction in the instruction set has a binary value, so machine code, which is written in binary, can be directly understood by the processor without translation.

Typical Operations:

- Data transfer
- Arithmetic calculations
- Comparison
- Logical operations
- Branch (conditional)
- Shift (multiplication)

A machine code *instruction* usually has two parts: the *opcode* and the *operand(s)*. The opcode corresponds to an instruction and the operand(s) are effectively arguments.

5.12 Addressing Modes

When constructing a machine code instruction, part of the opcode is the *addressing mode*. This defines how the arguments ought to be interpreted. There are two main types of addressing mode: *immediate* and *direct*.

- when immediate addressing is used, the value to be used in an operation is specified in the machine code instruction as a constant
- in direct addressing, the value to perform an operation on is stored in the address given by the arguments (memory or register)

5.13 Assembly Language

Here are my x86_64 assembly notes and examples: `https://github.com/alexander-neville/assembly`. AQA has its own instruction set, found here: `https://filestore.aqa.org.uk/resources/computing/AQA-75162-75172-ALI.PDF` and there is a simulator capable of running these instructions here: `https://peterhigginson.co.uk/AQA/`

5.13.1 Fibonacci Example

This is a good example of some AQA assembly. The program prints the first 10 Fibonacci numbers, using some basic operations and comparisons.

```
// initialise some variables

    MOV R0, #1 // current number
    MOV R1, #0 // previous number
    MOV R3, #0 // counter
```

```
LOOP:

    MOV R4, R0 // backup current number
    ADD R0, R0, R1 // find next number
    MOV R1, R4 // store previous number
    OUT R0, 4 // print current number
    ADD R3, R3, #1 //increment by 1
    CMP R3, #10
    BLT LOOP // repeat if not the 10th iteration
    HALT // end of program
```

5.13.2 Bit-wise Operations

A bitwise operation operates on each bit, irrespective of its value. To determine whether a binary number is odd or even a bitwise and operation can be used. The binary number undergoes *AND* with *000. . . 1*. This operation is shown bellow (a `-` means any value)

```
-------1
00000001
=
00000001 = 1


-------0
00000001
=
00000000 = 0
```

The result is only *1* if the last bit of the number is *1*. Other digits are always *0* after this operation. (anything and *0* is *0*)

To flip all the bits, a logical *NOT* operation can be used. Additionally a register can undergo an *XOR* operation with *111. . . 1* to achieve the same result. Once a number has been inverted, 1 can be added to find the two's complement of the original number.

5.13.3 Logical Shifts

During a shift operation, the entire contents of the register can be moved. In a left shift *(LSL)*, the *most significant bit* is moved out of the register. In a right shift (*LSR)*, the *least significant bit* is moved out of the register. The bit which is lost from the register is stored in the *carry flag* in the *status register*. This kind of operation can be used to check whether a number is even or odd:

```
01011011
00101101 ;; carry 1
01011010 ;; the lsb has been zeroed

01011010
00101101 ;; carry 0
01011010 ;; the result is the same
```

Following a *LSR* and then a *LSL* the least significant bit is set to zero, irrespective of its initial zero. The result of this operation can be compared to the initial value. If the two values are not equal, the *LSB* must have been a one and hence the initial value was odd.

5.14 IO Devices

5.14.1 Barcodes

A barcode is a reliable way of storing a small amount of information. This makes them suitable for storing some sort of identification number that can be looked up in a database. It is up to retailers to store information about the associated product. Eg. Two retailers may sell the same product, with the same barcode, although each retailer will store different data about that product, including price, etc.

Many different standards are used for encoding data in barcodes. The most common are the European Article Number *(EAN)*, sometimes called *IAN*, and other barcode standards recognised by *GS1*, a not-for-profit standards agency based in Belgium. Another common type is *code 128*, which can store characters and is often used in shipping and logistics. It is the standard of the barcode that determines its appearance and how it can store data.

Description:

A barcode can be described as *one dimensional*. The benefit of such a tall barcode with all the data arranged lengthways is reliability; the barcode can be accurately read even if part of the total height is damaged.

A barcode will typically include a *quiet area* before the barcode to reduce interference. In addition *guide bars* are found at the beginning, in the middle and at the end of the barcode *(EAN)*. This helps frame the barcode contents, making it easier to interpret. In order to reduce errors, the second half of a barcode is a copy of the first half, with dark and light areas inverted. A barcode may also contain a check-digit.

Scanners:

A barcode reader will emit laser light, which is reflected by a moving mirror over the whole barcode. The black strips on the white background reduce the light reflected from certain (black) parts of the barcode. The amount of laser light returned is detected by a photo-diode or a *CCD*, and is converted to an electrical signal. This undergoes *ADC* conversion and then the data can be retrieved from the bit pattern.

5.14.2 QR codes

A *quick response* code is a type of *two dimensional* barcode that can be read by smartphones and other personal devices. *QR* codes are able to store more information than a barcode, although more processing is needed. This means they are more suitable for storing complex data like *URLs*, rather than id numbers for use in an organisation. *QR* codes are inherently less reliable and less tolerant to damage than barcodes, because more information is packed into a smaller space, leaving much smaller margins for error.

Reading QR Codes:

Computer vision and image processing techniques are used to find the data encoded in a *QR* code. The photograph to work on is obtained with the device's camera. Bitwise logic can be used to check the areas of the barcode and determine if a pixel is light or dark.

5.14.3 Digital Cameras

A camera allows analogue data (light) to be converted to digital data and stored within a computer system. These can be purpose built devices, although it is common for mobile telephones to have a camera assembly.

Components:

- shutter
- lens
- colour filter
- sensor

Description:

When the shutter is open, light is focused onto the *sensor* by the camera's lens. The sensor might be a Charge Coupled Device *(CCD)* or a Complimentary Metal Oxide Semiconductor *(CMOS).* In either case, the intensity of light reaching the sensor is measured in millions of locations, by photoelectric cells (one for each pixel in image). *Colour Filters* are used to separate light into three channels ahead of the sensor, so colour can be recorded.

5.14.4 RFID

Radio Frequency Identification is a method of storing and transmitting small amounts of information over small distances via radio waves. The RFID system does not need line of sight, nor physical contact to transfer data. Different implementations of the system have ranges from a few cm to hundreds of metres.

Components:

- Receiver/Reader
- Transponder
- Microchip
- Antenna

Description:

RFID *tags* are often attached to inventory items like a barcode. Using a *passive* system, the RFID tag is brought near to a reader, which is emitting radio waves. When in range, the tag's antenna picks up the radio communication. The transfer of energy to the device activates the tag's *IC* chip, which modulates and returns an EM signal to the reader.

The *passive* system depends on high intensity emission from the reader to be activated, so the *transponder* (RFID device), must be close to the reader. *Active* systems have a power source, so they are able to transmit a signal to a receiver that is much further away.

5.14.5 Laser Printers

A laser printer is ideal for printing documents in large volumes. As opposed to the liquid ink in an *inkjet* printer, a laser printer uses dry, powdered toner. The up-front cost of a laser printer is high, although the running costs are often lower.

Components:

- toner hopper
- drum
- laser unit
- mirror
- heat fuser

Description:

Before a page is printed, the drum is covered in a negative electric charge (excess of electrons). The mirror assembly reflects the beam from the laser light source over the drum, removing the negative charge in certain areas and creating an inverse of the image to be printed. Negatively charged toner adheres to the positive/neutral parts of the drum. Paper is rolled across the drum and the toner is transferred to it, creating the image on the page. Finally the paper is passed through the fuser, where it is heated, binding the toner to the page.

5.15 Secondary Storage

Registers, cache and main memory are all *volatile* storage media and they depend on electrical power to hold data. In addition, the cost of *RAM* and other motherboard components per unit of storage is high. The physical space available on the motherboard and the processor chip is also limited.

These factors introduce the need for an alternative, *non-volatile*, mass storage media. The name given to this kind of storage is *secondary storage*. Secondary storage devices can store data without electrical power, so it is possible to store data across multiple boot cycles. Additionally, it is possible to manufacture these devices with large storage capacities at a relatively low price per unit of storage.

Secondary storage is more distant from the processor, so it can take a long time for data to be returned. As a result, the processor will never fetch instructions directly from secondary storage without loading them into memory first.

5.15.1 Hard Disk Drives

A *HDD* is a type of magnetic storage; ferrous iron particles can be polarised to encode data.

Terminology:

- disk
- platter
- sector
- read-write head

- spindle

Description:

A *HDD* has many platters, circular disks with top and bottom sides exposed attached to a central spindle. *Read-write heads* rest slightly above each surface. If there are four platters, each with a top and bottom side in close contact with a head, a whole byte can be read in parallel. Each surface has concentric rings, split into sectors containing many magnetised 'spots'. As the head is moved over a sector, a change in the magnetisation represents a *1*, while no change is equal to *0*.

Performance and Reliability:

To retrieve data from a hard disk drive, the read-write heads must be moved to the sector containing the data to be read. Actuators move the heads to the right ring, while the spindle is rotated quickly to move the sector under the heads. To improve seek time, the speed of the disk can be increased. A fast drive will spin as quickly as 10,000 rpm.

The use of moving parts can make this type of storage less reliable. It is possible that detritus in the drive can cause the disk to be damaged and the data corrupted at any time.

5.15.2 Optical Devices

Optical disks are a portable way of storing smaller amounts of data. Optical disks may be read only *(CD-ROM)*, recordable *(CD-R)* or fully re-writable *(CD-RW)*. Optical disks are cheap to manufacture and distribute. These disks can be removed from one device and moved to another with ease.

Terminology:

- pit
- land
- spiral track
- laser

Diagram:

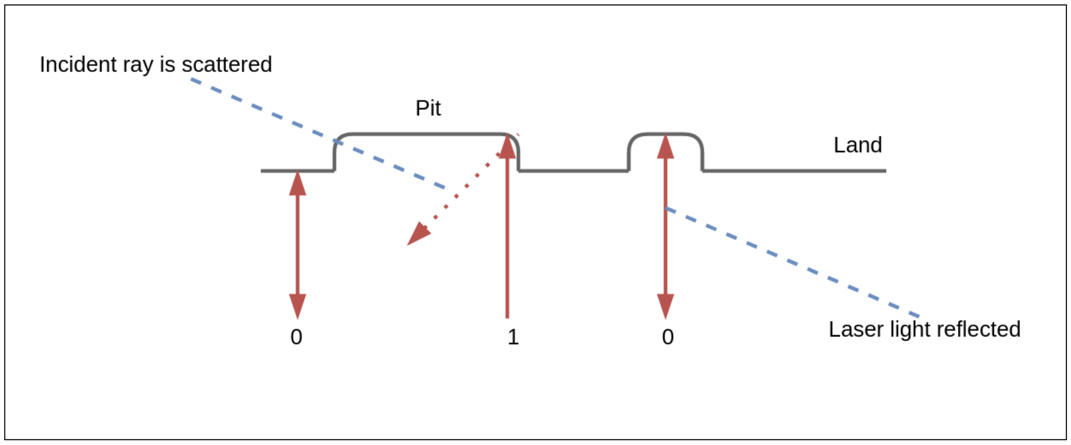

Figure 3: The pits and lands of an optical disk

Description:

During manufacturing, intense laser light is reflected onto a *CD-ROM* disk to burn pits along the *track*. During playback, a laser of lower intensity is directed at the track as it spins. At the beginning or end of a depression, light is scattered and not reflected back to the sensor. An area like this represents a *1*. When the laser falls on the middle of a pit or land, light is reflected back towards the sensor and a *0* is detected.

A recordable disk is covered with a transparent dye. A high intensity laser can alter the reflective properties of the dye. As the CD is read, the changes in the property of the surface affect the amount of light reflected, rather than pits and lands.

Certain types of compact disk can be rewritten. A high powered laser heats and deforms the surface of the disk. A magnet is used in conjunction with the laser to set the state of the spot while it is being heated. Similarly a *DVD-RW* uses a 'phase change alloy' which changes between *amorphous* and *crystalline* states under the power of the laser light.

Performance:

A typical *CD-ROM*, the oldest type of optical storage can hold about 650-700 MB of data, while a modern *Blu-Ray* disk can store upwards of 50 GB of data. As technology has improved, shorter wavelengths of light are used to read the disk. This means the size of pits and lands can be reduced and still read at the same apparent resolution. More pits and lands can fit on the same length of track and the spiral can be packed more closely, increasing the amount of track that can fit on a single disk.

5.15.3 Solid State Drives

A Solid State Drive *(SSD)* is a modern type of secondary storage, frequently used in personal devices. There are two common implementations of *EEPROM*, those which use *NOR* logic and those which use *NAND*. The latter technology is more widely used in mass storage, as the storage density is higher and the cost per unit of storage is lower.

Components:

- Page
- Block
- Control Gate
- Floating Gate
- Oxide Layer
- Bitline
- WordLine

Diagram:

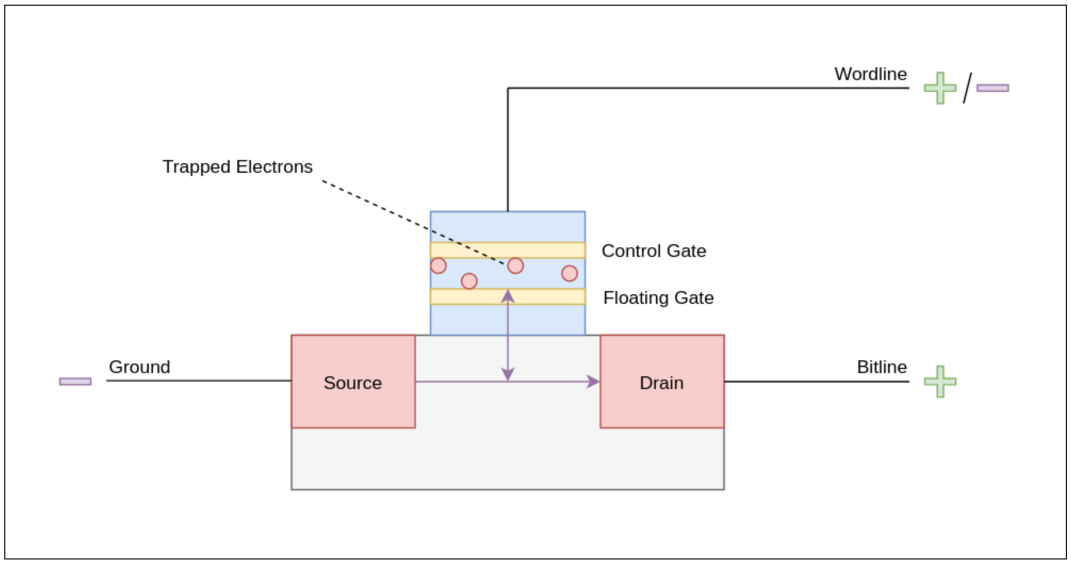

Figure 4: A nand cell

Description:

In order to retain information, a single nand cell contains two gates separated by *oxide layers* which electrons cannot usually cross. As the *bitline* is given a positive charge, electrons are drawn from the *source* to the drain.

- If the *wordline* is set to positive, some electrons are drawn up the oxide layers and trapped by the *floating gate*
- If the power is turned off, any trapped electrons remain in position
- If the *wordline* is set to negative, any electrons are forced out of the floating gate, clearing the cell

No charge in the trap is considered a *1*, while any trapped electrons register a *0*.

NAND memory divides storage into *pages* and *blocks*. It is not possible to overwrite existing pages using NAND technology and so a *block* must be erased entirely if its constituent pages need to be modified. While it is possible to write data to a single page, it is not possible to *erase* one page alone and so the whole block must be backed up and cleared.

Performance:

The typical capacity of an *SSD* is smaller than that of a *HDD* and the price is generally higher per unit of storage. Solid state devices have the advantage of no moving parts, reducing the electrical power required and the space needed to install an SSD. This makes them useful in small mobile devices like phones and tablets, where space and battery power are limited. In addition, solid state devices are faster to read and write from as no seeking time is required; read-write heads do not have to be moved to a certain location before data can be read or written.

While the longevity of *SSDs* is a topic of debate, they can be considered generally more reliable. The absence of moving parts reduces of the chance of the device suddenly failing, although the number of read write cycles is limited.

6 Data Structures

not implemented

7 Algorithms

not implemented

8 Regular languages

A *regular language* can be defined by a regular expression or finite state machine.

8.1 Mealy Machines

Mealy machines are a type of FSM with an output determined by the current state and the input. The example in figure 5, is an exclusive OR operation on the last two inputs.

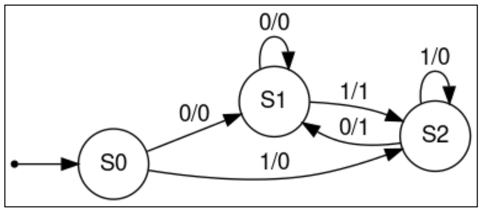

Figure 5: A typical Mealy machine

The state transition diagram can also be represented with a transition table.

Input	Current	Ouptut	Next
0	S0	0	S1
1	S0	0	S2
0	S1	0	S1
1	S1	1	S2
0	S2	1	S1
1	S2	0	S2

8.2 Sets

A set is an unordered collection of values, in which any one value may occur at most once. Sets may defined simply by listing each member.

$$A = \{2, 4, 6, 8\}$$

There are a number of common sets:

- empty set: $\{\}$ or ϕ, containing no elements
- natural numbers: $\mathbb{N} = \{1, 2, 3, 4, 5, ...\}$
- whole numbers: $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$
- rational numbers: $\mathbb{Q}$, any value which can be expressed as a ratio
- real numbers $= \mathbb{R}$, the set of real world quantities

8.2.1 Finite and Infinite Sets

A finite set contains a number of elements which can be counted off to a particular number. Finite sets include $\{1, 6, 8, 9\}$ and $\{1, 3, 5, ...99\}$. The *cardinality* of a set is the total number of elements in the set. Infinite sets may countable or not countable. Despite being infinite, $\mathbb{N}$ is countable; it is possible to determine the next number in the set and count of elements. A fully countable set can be measured against a subset of the natural numbers, whereas a countably infinite set, can be countered interminably.

8.2.2 Set Comprehensions

More complicated sets may be defined by comprehension, a more compact syntax than writing out the whole set.

$$B = \{n^2 | n \in \mathbb{N} \land n < 5\}$$

8.2.3 Cartesian Product of Two Sets

The *Cartesian product* of two sets is written $A \times B$ and is the set of all ordered pairs (a, b) where a appears in A and b appears in B. If $A = \{2, 4, 6\}$ and $B = \{1, 3, 5\}$, $A \times B$ is:

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

8.2.4 Sets and Subsets

If every element in A is also a member of set B, A is a subset of B, written:

$$A \subseteq B$$

It could also be said that B is a superset of A:

$$B \supseteq A$$

If B contains another element not in A, then A is said to be a proper or strict subset of B:

$$A \subset B$$

8.2.5 Set Operations

The *union* of A and B is all of the elements in either set or in both, written:

$$A \cup B$$

The *intersection* of A and B is all of the elements both sets, written:

$$A \cap B$$

The difference between the sets A and B is all the elements in one set, but not the other, expressed:

$$A - B = \{x | x \in A \land x \notin B\}$$

8.3 Regular Expressions

A regular expression is a string of characters used as a search pattern in text. Regular expressions can be used to define the string accepted by an automata. Various metacharacters have special significance in regular expressions. These are summarised in the table below.

Metacharacter	Use
()	Grouping
Pipe	Logical OR
?	0 or 1 occurence
*	0 or more occurrences
+	1 or more occurrences

8.4 Turing Machine

A Turing machine is an ideal computer system which is possible of implementing any computer algorithm. The machine has a *head* capable of reading and writing to cells on an infinitely long piece of tape. During operation, the system is capable of performing an operation determined by its current state; the contents of the current cell and a set of user-specified instructions. A Turing machine may operate on any alphabet.

The state transition diagram of a Turing machine designed to increment a binary number by one is show below. The equivalent finite state machine is show in figure 6.

Current	Input	Output	Dir.	Next
S0	E	E	L	S1
S0	0	0	R	S0
S0	1	1	R	S0
S1	E	1	R	S2
S1	0	1	L	S2
S1	1	0	L	S1
S2	E	E	L	S3
S2	0	0	R	S2
S2	1	1	R	S2

For any particular row in the state transition table, a *transition function* can express the behaviour of the machine. The syntax is given below.

$$\delta(\text{Current, Input}) = (\text{Next, Output, Dir.})$$

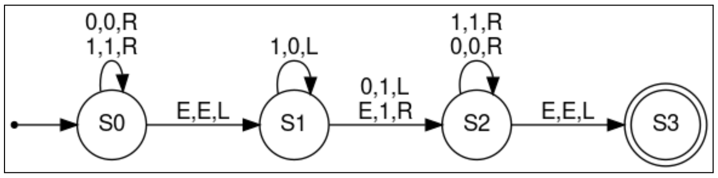

Figure 6: A typical Turing machine

This simple type of Turing machine is capable of performing one type of computation. The *Universal Turing Machine* is capable of simulating any commutable sequence. *"if this machine **U** is supplied with the tape on the beginning of which is the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**"*. If **U** was able to interpret the description of **M** in this way, this constitutes a very abstract description of the *stored program computer*.

8.5 Backus-Naur Form

Spoken languages, used to communicate between people, are called *natural languages*. These languages are loosely defined and too ambiguous for computers to interpret. Programming languages, which are designed to be interpreted by a computer, are called *context-free grammars*, which means that any non-terminal string in a production rule can always be expressed as a string of terminals and/or non-terminals, as in:

$$A \to \alpha$$

A *terminal* is an elementary piece of syntax in a grammar, for example a digit or letter; it cannot be expressed by any other combination of symbols. *Context-free* means it is not significant where the production rule is applied; Any non-terminal on the left can be expressed in terms of the string of non-terminals and terminals on the right.

The definition of a grammar is called a *meta-language* or *meta-syntax*. A common example is *Backus-Naur Form* (BNF). Such a syntax is capable of many constructs a regular expression cannot handle and is able to express some others far more elegantly. Features like recursion are made easy in BNF: constructs like numbers may be defined as either a single digit, or an existing number and another digit. The process of applying the grammar rules to a string to determine validity is called *parsing*. BNF syntax is a set of grammatical rules, each of them called a *production*

```
<expression> ::= <factor> | <factor> * <factor> | <factor> / <factor
<factor> ::= <term> | <term> + <term> | <term> - <term>
<term> ::= <expression> | <number>
<number> ::= <digit> | <digit><number>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

9 Communication Technology and Consequences

9.1 Communication Methods

9.1.1 Serial Transmission

During serial data transmission, bits are set one after the other down a single (serial) data channel. A second data wire will be needed for simultaneous bi-directional data transfer. Sometimes an additional *ground* wire is used to reduce the adverse affects of interference. Additional control wires may also be used in serial data connections.

9.1.2 Parallel Transmission

Multiple bits of data are sent simultaneously along a number of parallel data wires.

9.1.3 Comparison of Methods

Each of the wires used to transmit individual bits in a parallel connection will differ slightly from the others. This subtle difference will affect the rate at which data can travel along the wire and hence the time taken for a bit to travel a certain distance. This causes bits to arrive at the receiver at different times, a phenomena know as *skew*.

Parallel transmission becomes impractical over larger distances, as *skew* becomes more pronounced. Parallel transmission is, however, suitable in some environments, for example onboard motherboards and *Integrated Circuits*, where distances are small and the increased speed is valuable.

Serial connections have much smaller interfaces, making them suitable for mass manufacturing, especially in mobile devices and consumer electronics.

Crosstalk between the data channels of a parallel connection can cause interference and corruption. The danger of interference due to crosstalk increases with frequency. As a result, the frequency of serial connections can be safely increased beyond the practical limits of parallel connections, meaning more data can be transmitted in a given amount of time, even if less data is transferred per cycle.

9.1.4 Bit and Baud Rate

The two measures are linked with this equation:

*bit rate = baud rate * number of bits per baud*

In typical serial connections, *1s* and *0s* are represented by high and low voltages in a cable, called non-return-to-zero *(NRZ)* communication, where the signal voltage never returns to *0v*. The *baud rate* is the number of symbol changes per second - the number of times the signal voltage is changed / the frequency. The bit rate may be the same as the *baud rate*, although - using signal modulation - more than 2 values may be encoded within a single cycle. In such a case, the *bit rate* (the total number of bits transferred in a second) is equal to the *baud rate* * the number of bits per *baud*.

Increasing the number of bits per *baud* means more data can be transferred along a narrower (possibly serial) cable in a certain amount of time.

9.1.5 Baseband and Broadband

Broadband is an analogue data transfer method, meaning there are a *continuous* range of values (voltages) and each symbol change can represent more than two values (as in binary). Broadband connections are often multi-channel and bidirectional. These connections are frequently use in *WANs*.

Baseband is a type of digital data connection, commonly used in *LANs*, where the bit rate is often the same as the baud rate, hence each symbol is either a 0 or 1. Ethernet cables are baseband connections.

9.1.6 Latency and Bandwidth

Latency is the time taken for some data to be transmitted a certain distance, ie. from sender to receiver, irrespective of how much data is sent at once. Latency is often measured in seconds *(s)* or milliseconds *(ms)*.

Bandwidth loosely means how much data can be received at once, irrespective of the speed the signal can travel and hence the latency between sender and receiver. Bandwidth is often measured in bits per second *(bps)* and has a direct relationship to bit rate.

In data communication and networking, bandwidth is analogous to data transfer rate.

9.1.7 Synchronous Communication

During *synchronous transmission*, both sender and receiver share a common clock cycle for coordinating signals. This communication method depends on that clock cycle to govern communication along the wire. This type of communication is best for connections that work in real time, with a constant flow of data, for example within a computer's processor.

9.1.8 Asynchronous Communication

There is another common communication method known as *asynchronous transmission.* Neither the sender, nor the receiver share the same clock cycles. Communication is governed by start and stop bits, sent before and after a communique. The stop bit is always the opposite of the start bit. Despite the absence of a common clock, the sender and receiver must use the same baud/bit rate so that the receiver can understand the message once the start bit is received.

9.2 Network Topology

A single computer with no connection to any other devices is described as a *stand-alone* computer system. When a computer is connected to one or more other computers, the resulting system can be described as a *computer network*. Digital *baseband* connections are often used inside a *local area network*. Many smaller LANs spread over a large area are often joined to others by *broadband* connections, creating a *wide area network*.

Physical network topology defines how the devices are physically connected with hardware devices and equipment. *Logical* network topology is the layout used by devices on a network to communicate using the physical network equipment provided.

9.2.1 Physical Bus Topology

The *physical bus topology* is a simple network configuration, used in many small home networks. Each device is connected to a *backbone* cable which runs past every device. A *terminator* is placed at either end of the bus.

Advantages:

- A bus network is inexpensive to set up and easy to maintain.
- Less cable has to be laid/installed.
- Identifying problems with equipment can be easier.

Disadvantages:

- Data intended for one device on the network passes many other computers.
- The single backbone cable is subject to congestion as many devices need to communicate through the network.
- Collisions can occur as devices need to send data along on the bus.
- The single backbone cable is a single point of failure. If it is not functioning, it is impossible for devices on the network to communicate with one another.

9.2.2 Physical Star Topology

The *physical star* network configuration is a more complicated type of network, where each node has its own dedicated connection to the hub or router.

Advantages:

- The star network is more secure, as data intended for one computer on the network does not pass by others.
- Dedicated cables for each device eliminate the risk of collisions between signals sent by different computers.
- This type of network is more flexible and expandable, as more hubs and devices can be connected to the hub.

Disadvantages:

- Star networks can be more difficult and expensive to implement as more cables need to be laid/installed.
- The central hub is a single point of failure. Should the hub fail, none of the devices on the network will be able to communicate with one another.

9.2.3 Logical Topology

Logical topology is the manner in which data is handled on top of a physical network. For example, a hub might use a bus protocol to push data outward onto a star network, similar to how a physical bus network behaves.

9.2.4 MAC Addresses

9.3 Client/Server

A *server* is a computer which services requests from a number of clients; a response is sent back to the connected client in return. A computer may be both a client and a server simultaneously. A server might be used to process or store data.

Within a LAN, server(s) may be set-up to handle tasks common among all client computers on the network. Internet facing *web-servers* are used to host websites and content on the internet.

Advantages:

A client-server configuration is preferred when central management over the whole network is needed, making this model popular in schools and businesses. Servers might be set up to handle user accounts, store files and manage backups.

Disadvantages:

A client-server network requires expensive hardware (the servers themselves) and personnel to maintain the servers. This makes the client-server model impractical on smaller home networks.

9.4 Peer-to-peer

There is less core infrastructure in a decentralised peer-to-peer network. The services that would be provided by servers are shared amongst the clients.

Advantages:

No expensive server computers are required. Expertise and maintenance are not required on a peer-to-peer network.

Disadvantages:

All clients must be connected and powered-on for the network to function as expected. Peer-to-peer networking may leave a user's files visible to other devices on the network.

9.5 Wireless Networking

9.5.1 Wi-Fi

Wi-Fi, standing for *wireless fidelity*, is a type of wireless network standard designed to be interoperable with *IEEE 802.11* protocol and work alongside Ethernet at the *Network Access* layer. Devices using wi-fi can connect to a *wireless access point* and communicate with any other device on the network.

9.5.2 NIC

In order to connect to a wireless network, a device must have a wireless network interface card *(NIC)*, a device will have a similar card for all its other interfaces, eg. Ethernet. The NIC has a hardcoded MAC address. The combination of an NIC and a computer is called a *station*.

9.5.3 SSID

A service set identifier *(SSID)* is a human readable name for a wireless network. It may be broadcast to devices within range of a *WAP*, or kept private.

9.5.4 Security

Any device within range of a *WAP* could connect to an unprotected wireless network. A network password is often used in wi-fi protected access *(WPA)* networks. *WPA2* is also a common standard. A network owner may also choose to set up a MAC whitelist for ultimate control over which devices may connect. In order to connect, the MAC address of a computer's NIC must be added to the whitelist.

9.5.5 CSMA/CA

Connected devices share the same channel to transmit data to the *WAP*. In order to prevent multiple computers trying to communicate with the *WAP* simultaneously, Carrier Sense Multiple Access with Collision Avoidance *(CSMA/CA)* is used. Before data may be sent to the *WAP*, the station checks if the channel is idle. If another device is communicating over the channel, the station waits a random amount of time before checking the status of the channel again. This process continues until the channel is free and the station is able to send data to the *WAP*.

9.5.6 RTS/CTS

One of the shortcomings of the *CSMA/CA* standard is the *hidden node* problem. It is possible that the *WAP* is engaged with a station that cannot be seen or heard by a station that needs to send data. This situation is common on larger networks, where the *WAP* serves a larger area.

Once the channel appears idle to a station, a 'request to send' signal is sent to the *WAP*. If the 'clear to send' signal is not received, the station waits a random amount of time before checking the channel status and resending the 'request to send' signal. If the *WAP* is free, a 'clear to send' signal is returned to the station and the data can be transmitted.

9.6 Communication & Privacy

not implemented

9.7 Social, Cultural & Legal Issues

not implemented

10 The Internet

The *Internet Protocol Suite*, often referred to as the *TCP/IP* stack, is a collection of communication protocols adopted on the public internet and other similar computer networks. The whole protocol suite governs end-to-end communication between devices including stipulations about the way data is split into packets, addressed transmitted and routed. Much like the *Open Systems Interconnection*, these conceptual network models arrange communication protocols into abstract layers.

10.1 Uniform Resource Locators

A URL is a specific type of *Uniform Resource Identifier* or URI, designed to identify web resources on computer networks. A typical URL may resemble:

```
https://www.example.com:443/path?query
```

In the example above the **https** protocol is used, followed by the characters `://`. This whole component of the URL is called the *scheme*. The next component is the host name or IP address, separated from the port number by a colon. The remaining components are the path to the resource or file in question and the query string, separated by a question mark.

10.2 Domain Names & DNS

A domain is an identification string which defines or identifies a *"realm of administrative autonomy, authority or control"* within the internet (Wikipedia). Domain names conform to the rules and regulations of the *Domain Name System*, which maps a domain to other types of information, such as an IP address. Any such string registered in the **DNS** is considered a domain name.

Domain names are organised, or grouped, in subordinate divisions of the domain root, which is nameless (refer to **FQDN**). The first level of domains are called *top-level-domains* or **TLDs** for short. **TLDs** include generic top-level domains such as `com`, `info`, `net`, `edu`, and `org` as well as the country code top-level domains. Below this tier are the second and third-level domains. These are openly available for reservation to internet users. Domain names work from right to left, with the top-level domain appearing on the right. Domain levels are separated with the `.` (full stop) character.

- A *fully qualified domain name* is all the components of a domain name, which collectively specify the domain's exact location in the domain name system. **FQDNs** are terminated with a full stop, which is characteristic of this type of domain. The point represents the root domain, from which the domain can be located.
- A *subdomain* is federated by the owner of the parent domain. There is no limit to how many subdomains may be used. Third-level domains are often used to identify a particular host server, eg *mail* or *www*. Such subdomains may only support the implied functionality.

The *Internet Corporation for Assigned Names and Numbers* (ICANN) is the governing body for the name and number systems used on the internet. There are five *Regional Internet Registries* globally. Registries are organisations which manage top-level domains. Through a memorandum of understanding, these registries cooperate through the unincorporated *Number Resource Organisation.* Internet resources are distributed to RIRs and disseminated in accordance with the policies of the registry. The commercial sales of domain names are delegated to accredited *internet registrars.* When a customer purchases, or more accurately *leases*, a domain, the registrar notifies the registry which maintains the records. Regional registries allocate blocks of IP addresses to local internet registries, most often internet service providers.

10.3 Protocol Stack

The internet protocol suite is a layered model, in which transmitted data descends through the layers as it is sent. This process is reversed when the data is received. At each layer, the data is further encapsulated in accordance with the workings of the lower-level protocol. The data handled at each layer is sometimes referred to as the *Protocol Datagram Unit* (PDU). A PDU is composed of protocol-specific control information or header and trailer sections and user data, which may be further encapsulated by protocols operating in layers above. Each layer communicates with the layers immediately adjacent to itself, either the layer above or below during operation.

- **Application Layer:** contains abstract communication protocols used in process-to-process communication over an Internet Protocol (IP) network.
- **Transport Layer:** responsible for device-to-device (or end-to-end) communication for applications. This layer facilitates connection-orientated communication (TCP) or best-effort stateless communication (UDP). The PDU for the transport layer is the *segment* for TCP and the *datagram* for UDP.
- **Network Layer:** responsible for inter-network communication through network gateways. The PDU of the network layer is the IP packet. The protocols working at this level, most notably IP, are not connection orientated themselves, although can respond to transport layer service requests to provide such a system.
- **Link Layer:** communication protocols limited to the physical connections of the current node within the same network segment. The link layer PDU is called a *frame*.

10.4 IP Addresses

An *Internet Protocol* address is a unique number assigned to a network interface involved in internet communication. The address identifies and locates the host, which allows a path to the host to be established. IPv4 is a 32-bit standard, although such small addresses have caused IP address shortages. To address this IPv6 addresses are 128 bits in length, offering many more addresses. A single IP address must be unique within a particular network, though the same address may be used on two separate networks.

The first portion of an address refers to the network, while the remaining bits are used to identify specific devices. Classful addressing was used to separate the network and host sections of a given address. The table below summarises how many bits were used in class A, B and C addressing. Class D was used for multicast addressing and class E was used for reserved addresses.

Class	Network Bits
A	8
B	16
C	24

Classful addressing has been superseded by CIDR notation, where the number of bits in the host portion is appended onto the IP address after a forward slash, for example: `103.27.104.92/24`. This value can be used to determine of the subnet mask, which can be applied to the IP address with a logical AND operation to reveal the network portion. The subnet mask is a 32 bit number containing 1s in what would be the network portion of the address.

Large networks are often divided into smaller subnets for ease of operation and performance. Administration is made easier by reducing the broadcast domain of a single device on a network.

10.5 TCP Handshake

Transmission control protocol is used in conjunction with IP, which is not stateful by itself. TCP is solely responsibly for managing a stateful connection. The journey of each segment is managed by the lower level IP protocol, while TCP remains responsible for the overall dialogue; reconstructing whole communications from individual segments and requesting lost packets are sent again. A TCP segment consists of a segment header and data section.

To open a TCP connection the sender and receiver engage in a three way handshake.

1. **SYN:** sent from the client to the server, the first communication instructs the server to synchronise its sequence number with the number generated on the sender side. Other information including the *Maximum Segment Size* (MSS) and the *window size* or buffer capacity of the sender are also sent.
2. **SYN + ACK:** receiver replies with its own sequence number and a synchronisation flag of 1, in addition to an acknowledgement flag. The MSS and window size of the receiver are also sent. Both parties agree upon the smaller MSS to avoid packet fragmentation. From information exchanged at this point, the number of segments that can be exchanged in either direction can be calculated at each end. Finally an acknowledgement number is transmitted, which is the original sequence number incremented by 1.
3. **ACK:** sender replies with the sequence number received as the acknowledgement number from the receiver. The transmitted acknowledgement number is the synchronisation number of the receiver incremented by one and the acknowledgement flag is set to 1.

There are two disconnect methods: the abrupt TCP reset (RST) and the graceful four way disconnect. The RST is sent when either TCP entity is obliged to close the connection or either entity closes both directions of data transfer. The four way disconnect occurs as follows:

1. **FIN:** whichever party initiates the termination sends a TCP segment with the FIN bit sent to 1. The initiator will enter the **FIN_WAIT_1** state, waiting for acknowledgement.
2. **ACK:** acknowledgement in sent immediately from the receiver to the initiator. When the initiator receives acknowledgement from the receiver, it enters **FIN_WAIT_2** state, until a TCP segment containing a **FIN** bit is received.
3. **FIN:** after some closing operations are completed by the receiver, another TCP segment is sent, with the **FIN** bit set.
4. **ACK:** the initiator enters the **TIME_WAIT** state and final acknowledgement is sent. The acknowledgement may be resent during the period the initiator remains in the waiting state. After a certain implementation dependent time, the connection is formally closed.

10.6 Asymmetric Encryption

Asymmetric encryption is a vitally important component of internet communication. Using *public key cryptography* it is possible to establish fully encrypted bidirectional communication without sharing a common key in advance. A cryptographic algorithm is required to generate mathematically linked public/private key pairs. Of the two, the public key can be widely disseminated, so long as the private key remains secret. Any party is able to encrypt a piece of information with the public key, which can only be decrypted with the private key.

10.7 Digital Signatures & Certificates

A *digital signature* is a methods of determining authenticity. If a client verifies a signature it is safe to conclude that the message was authored by a known sender. Signatures also offer *non-repudiation*, which means it can not be claimed that both a signature was not produced by a known party and the private key remains private. Digital signatures function as outlined below:

1. A key pair is generated by an appropriate key generation algorithm.
2. Signature is produced from the private key and the plain text.
3. Plain text, public key and signature are used in conjunction for verification.

To produce the signature, the original message may be encrypted in its entirety or a hash may be used in its place. Whichever is the case, the same procedure must be used on during verification.

A signature is an effective method of guaranteeing the authenticity of a particular message. To identify a particular host or server a *digital certificate* is used as well. These are issued by a trusted *certificate authority* (CA). A certificate will contain information such as the serial number, expiry date, holder credentials, holder public key and the digital signature of the CA itself. A CA certificate enables a host to operate secure, encrypted *TLS* connections.

10.8 TLS

Transport Layer Security (TLS) is an encryption protocol designed to secure internet communication. TLS and the deprecated SSL protocol which it replaces, do not fit neatly into any one layer of the TCP/IP stack. TLS depends on a stateful connection, so it exists somewhere above the transport layer. Like TCP, operating a TLS connections requires a handshake to establish a secure connection. The steps are outlined below:

1. Client initiates handshake, communicating information including the supported TLS versions and available cipher suites.
2. Server notifies client of selected cipher suite and provides digital certificate.
3. Client verifies the received certificate. The client generates a *pre-master secret* and encrypts it with the server's public key to be transmitted.
4. Using its private key, the server decrypts the incoming secret, which is now shared by both computers. This is used to generate a session key.
5. Fully symmetric bidirectional encrypted communication is achieved.

10.9 Local Networks

The rapid consumption of available addresses that afflicts the 32-bit IPv4 system has been slowed by the uptake of private, non-routable addresses in local networks. These addresses must remain unique within the local network, although there is no expectation that these addresses are globally unique, they may be reused on another network. As a result, acquiring a private IP address does not involve an internet registry. Communication between two devices on different networks requires another system, called *Network Address Translation* (NAT).

- *Dynamic Host Configuration Protocol* automatically provides IP information to devices on a network. This includes the allocation of an IP address and networking information such as the subnet mask. IP addresses may be allocated *dynamically*, whereby one device may receive any address from the pool of available addresses and be subject to a changing address, or by *static* reservation, where a single device permanently uses the same private IP address.
- If a device inside one network with a private IP address needs to communicate with a device on the public internet, its private IP address is replaced with the public IP address of the network gateway. Any response to the outgoing communication will reach the network router, which recorded ongoing communications in a translation table. Using this information, the publicly routable address is replaced with the private IP address of the correct device.
- To allow devices outside a particular network access to a server operating within, *port forwarding* rules are configured on the network router. The firewall maps incoming requests on specified ports to devices within the network.

10.10 Internet Security

Routing devices, joining two networks may incorporate a security system called a *firewall*. Different firewall implementations may be hardware or software devices. A dedicated firewall device may contain two NICs connected to a single computer, one facing the internal network that the firewall is designed to protect and the other facing the external untrustworthy network.

- *Firewalls* may use *static filtering* pre-configured rules, inspecting the source and destination IP address and port number of incoming or outgoing communications to determine if the traffic can pass.
- *Stateful Inspection* may be used to examine the payload contents of IP packets. This information is sometimes used to create temporary contextual rules for ongoing communication.
- *Proxy servers* are intermediary devices, offering protection to client devices, by concealing their identity. Source IP addresses are replaced with that of the proxy-server. When a response is received, the communications is forwarded to the original client. Proxy servers can operate a local cache, for frequently requested data.

11 Databases and Software Development

11.1 Introduction to Databases

The purpose of any database is to store data in the most intuitive manner possible, minimising the required disk space and delivering the most performant system. The Relational Database Management System *(RDBMS)* has become a popular database paradigm, although some others exist. A *RDBMS* is based on entity modelling and the relationships between entities.

11.2 Database Systems Overview

In production, applications like *mysql* or *mariadb* are used. These are two examples of a *RDBMS.* A database server tends to run as a service in the background. It is usually possible to connect to a running database server, either locally or remotely, through the database console. Within the *container database*, the individual databases are found. Each database is composed of *tables* and information can be retrieved from one or many of these tables during a *query.*

In the *RDBMS* paradigm, each *entity* within a dataset is represented with a *table.* A table will have pre-defined *columns*, one for each *attribute* an entity has. Rows in a table are called *records*, where one record represents one instance of an entity within a dataset. Each record has many *fields*, filled with the data corresponding to each *attribute* of the entity.

Every record in a table must be uniquely identifiable. A selection of columns, used together, will constitute the *primary key* (all the data needed to select exactly one record from a table). It is possible that a record can be identified with the value of just one column. When the values held by multiple columns are needed, this is described as a *composite key.* On an entity level, the information needed to identify an instance is called the *entity identifier.*

11.3 Relationships

Not only do entities have attributes, they also have relationships with other entities. Sometimes these relationships are logical, apparent within the data, while other times they are created to model data within the constraints of a *RDMS.* When such an entity is created it can usually be traced back to some abstract entity like a seat on a flight, a job listing or a sale in a shop.

The three types of relationships are:

- *one-to-one*
- *one-to-many*
- *many-to-many*

When one record maps to more than one other record, this is a *one-to-many* relationship. The term *many-to-one* is never used as its meaning is similar to *one-to-many* while relationships expressed like *many-to-one* rarely make sense in the real world.

A *many-to-many* relationship can often be problematic, especially in a *RDMS.* in order to reduce repetition within a database, a transaction table is usually built between tables with such a relationship. This reduces the *many-to-many* into two *one-to-many* relationships. The resulting transaction table will usually have a composite key, constituted by the foreign keys of the two records that are being linked. Seeing as this is the intended purpose of the table, a separate identifier is not usually required.

11.4 Normalisation

In order to reduce data duplication and make insightful queries easier to construct, there are some rules to follow. These rules dictate which *normal form* a database is in. *First normal form* is essential for most Database Management Systems *DBMS*, while third normal form is ideal. The general rule is:

The data depends on the key (1nf), the whole key (2nf) and nothing but the key (3nf)

11.4.1 Un-normalised

There are many companies involved in the production of a modern airliner, this hypothetical dataset joins aircraft to the companies which make the engines (not on specification!).

This table shows some data which is human-readable, but difficult to query. A major problem with this layout is the duplication of data. In a production database, this would significantly increase the size of the database and adversely affect performance. Data inconsistency is also a problem when the same data is stored more than once. In this table some fields have more than one value, which means the records are not uniquely addressable.

id	**name**	**price**	**engine _id**	**engine _name**	**engine_ quantity**	**supplier _id**	**supplier _name**
123	A320	100,000,000	1100G, 1A	geared turbofan, high bypass engine	2, 2	PW, CFM	Pratt & Whitney, CFM Interna-tional
243	737	100,000,000	1100G, 1B	geared turbofan, high bypass engine	2, 2	PW, CFM	Pratt & Whitney, CFM Interna-tional
156	A380	400,000,000	900, 7000	Trent 900, EA GP7000	4, 4	RR, EA	Rolls Royce, Engine Alliance
457	Typhoon	110,000,000	2000	EJ2000	2, 2	RR	Rolls Royce

11.4.2 First Normal Form

To achieve *first normal form* each field must have only one *atomic* value and each record must be unique. Each record also requires a unique key, whether that be composite or individual. In this case, the primary key for each row in the table is `id + component_id`. The table below shows all the data from before in *1nf*.

id	**name**	**price**	**engine _id**	**engine _name**	**engine_ quantity**	**supplier _id**	**supplier _name**
123	A320	100,000,000	1100G	geared turbofan	2	PW	Pratt & Whitney
123	A320	100,000,000	1A	high bypass engine	2	CFM	CFM Interna-tional
243	737	100,000,000	1100G	geared turbofan	2	PW	Pratt & Whitney
243	737	100,000,000	1B	high bypass engine	2	CFM	CFM Interna-tional
156	A380	400,000,000	900	Trent 900	4	RR	Rolls Royce
156	A380	400,000,000	7000	EA GP7000	4	EA	Engine Alliance
457	Typhoon	110,000,000	2000	EJ2000	2	RR	Rolls Royce

11.4.3 Second Normal Form

To satisfy *second normal form* the data must meet the criteria of *1nf* and there must be no partial dependencies. A partial dependency can happen when a composite key is used. To make *2nf* easy to achieve, relationships are often brought in here. To manage this, determine the separate entities in the data and create a table for each of them. By splitting up all the entities it is easier to make useful queries.

Relationships may be *one-to-one*, *one-to-many* (or vice-versa) or *many-to-many*. Where a *many-to-many* relationship exists, a transaction table is usually required, otherwise it is difficult to maintain the single field key needed for *2NF*. Unlike regular entities, a transaction table usually represents something abstract, like a sale in a shop, a job listing or a seat on a flight.

E.g. In this database, add a transaction table like this:

```
aircraft >----< engine

aircraft --< engine_option >-- engine
```

Here is how the entity tables will look:

aircraft

id	name	price
123	A320	100,000,000
243	737	100,000,000
156	A380	400,000,000
457	Typhoon	110,000,000

engine

id	name	supplier_id	supplier_name
1100G	geared turbofan	PW	Pratt & Whitney
1A	high bypass engine	CFM	CFM International
1B	high bypass engine	CFM	CFM International
900	Trent 900	RR	Rolls Royce
7000	EA GP7000	EA	Engine Alliance
2000	EJ2000	RR	Rolls Royce

engine_option:

This is the transaction table between the two tables. This table has a composite key of `aircraft_id + engine_id`. All the data in each record depends on the whole composite key. This table handles the multiple entries for each aircraft and engine, while avoiding partial dependencies.

aircraft_id	engine_id	engine_quantity
123	1100G	2
123	1A	2
243	1100G	2
243	1B	2
156	900	4
156	7000	4
457	2000	2

11.4.4 Third Normal Form

For data to be in *third normal form*, *1nf* and *2nf* need to be satisfied. In addition, data in a record may not have any *non-key* dependency, sometimes called lateral dependency. *2nf* already establishes the need to depend on the whole key, but *3nf* means that a field may not depend on any other attribute in addition to the primary key. In this database, the engine `supplier_name` depends on the engine's `id`, but it also depends on `supplier_id`. These situations are a good indication that another entity can be found and a new table created. This was not a problem in *2nf*, because a `supplier` has a less troublesome *one-to-many* relationship with the engine's `id`.

Here is the relationship that can be identified:

```
supplier ----< engine
```

With this type of relation, no transaction table is needed, therefore the complete *3nf* database looks like this:

aircraft:

id	name	price
123	A320	100,000,000
243	737	100,000,000
156	A380	400,000,000
457	Typhoon	110,000,000

supplier:

id	name
PW	Pratt & Whitney
CFM	CFM International
CFM	CFM International
RR	Rolls Royce
EA	Engine Alliance
RR	Rolls Royce

engine

id	name	supplier_id
1100G	geared turbofan	PW
1A	high bypass engine	CFM
1B	high bypass engine	CFM
900	Trent 900	RR
7000	EA GP7000	EA
2000	EJ2000	RR

engine_option:

aircraft_id	engine_id	engine_quantity
123	1100G	2
123	1A	2
243	1100G	2
243	1B	2
156	900	4
156	7000	4
457	2000	2

11.5 Diagrams

There are a number of different methods used to design and plan complicated database layouts. Some are visual, while others, like *entity descriptions*, are not.

11.5.1 Entity Descriptions

In an entity description, each line represents a table. The first section is the table name and then, within brackets, the attributes being implemented as columns. This vaguely resembles a SQL statement, although the data-types and constraints are missing. The key is underlined and any foreign keys are usually italicised.

Example from Chat Application:

- user (_id_, username, password, salt, last_login)
- chat_user (_user_id, chat_room_id_)
- chat_room (_id_, name, pin)
- message (_id_, text, time_stamp, *owner_id*, *chat_room_id*)
- attachment (_id_, path, time_stamp, *message_id*)

11.5.2 Entity Relation Diagrams

ER diagrams are a more visual way of representing a database layout. There are two common types, those which include the table/entity name only and those which show the columns and foreign key relationships. Here are some examples:

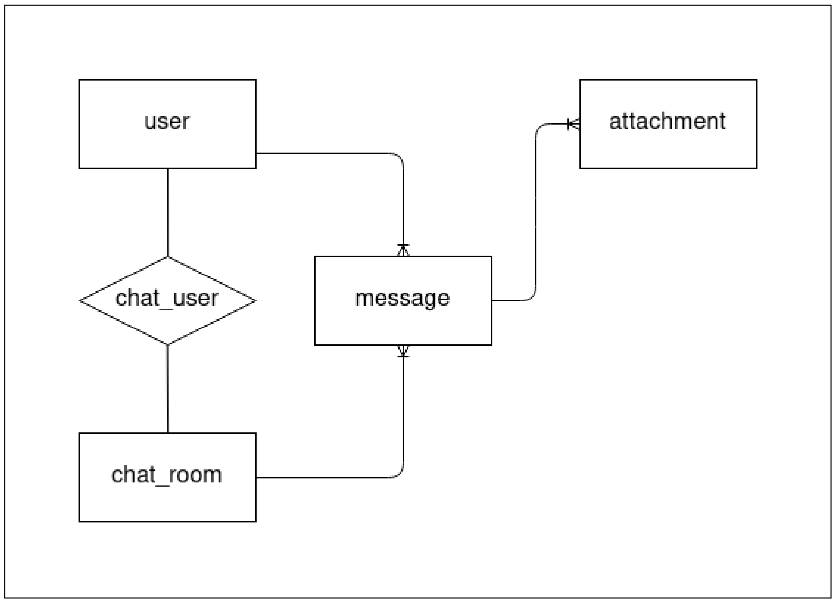

Figure 7: An abstract entity relation diagram

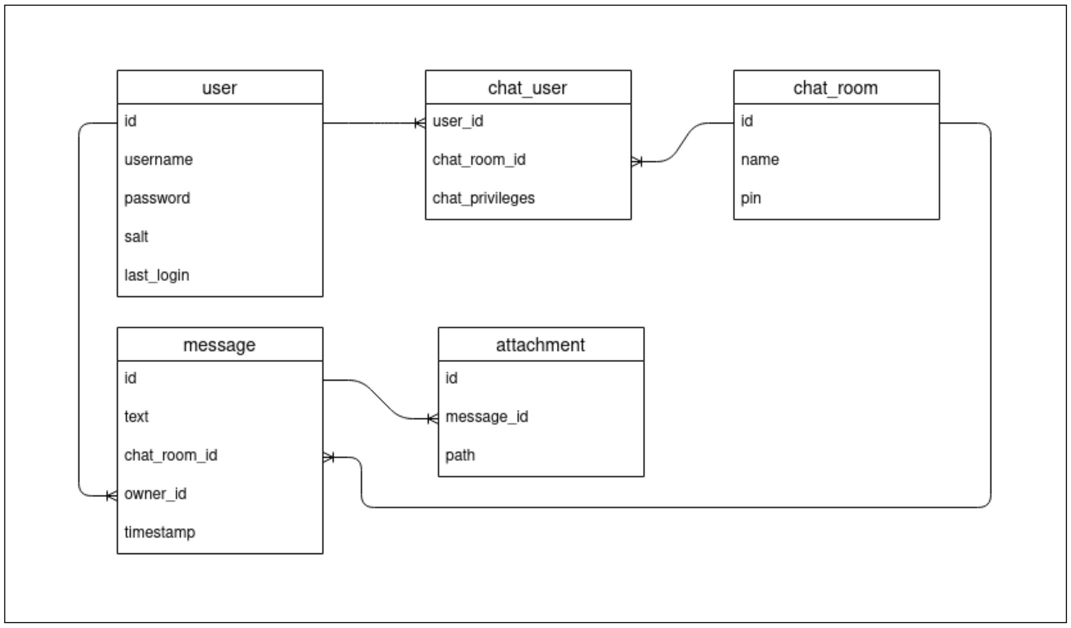

Figure 8: A more detailed entity relation diagram

11.6 SQL

SQL, standing for *structured query language*, is a declarative, high-level language for manipulating and querying databases. There are a number of SQL compatible database programs, two common ones being MYSQL and Mariadb. Database administration and set-up are not required parts of the A-Level specification.

SQL commands are generally shown in uppercase, although in most implementations they are case-insensitive. A command can span multiple lines, white-space rarely matters and commands are finished with a semicolon.

11.6.1 Creating Tables

Assuming that a database has been set-up, the first step is to create the tables which will hold the data. As there are some foreign key relationships, it is important that the tables are created in the right order.

This example will use the db design from the normalisation example above, leaving some mistakes to be corrected later.

```
CREATE TABLE aircraft (
    id INT(255) NOT NULL AUTO_INCREMENT,
    name VARCHAR(255),
    price FLOAT(24),
    PRIMARY KEY (id));

CREATE TABLE supplier (
    id VARCHAR(3) NOT NULL,
    name VARCHAR(255),
    PRIMARY KEY (id));

CREATE TABLE engine (
    id VARCHAR(10) NOT NULL,
    name VARCHAR(255),
    supplier_id VARCHAR(3),
    PRIMARY KEY (id));

CREATE TABLE engine_option (
    aircraft_id INT(255) NOT NULL,
    engine_id VARCHAR(10),
    FOREIGN KEY (aircraft_id) REFERENCES aircraft (id)
    ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (engine_id) REFERENCES engine (id)
    ON DELETE CASCADE ON UPDATE CASCADE);
```

Notice that each column has a datatype, the number following each one represents the maximum display width or the precision in the case of `FLOAT`. The best information on these data-types is the official documentation: `https://dev.mysql.com/doc/refman/8.0/en/data-types.html`

Extra Info:

- Features like *not null* and *auto_increment* are often used. Generally most columns are filled, although a column marked with *not null* will cause an error if no value is assigned. The id column of one of the tables is set to *auto_increment*, so a unique id is generated if one is not explicitly provided.
- Key constraints are used to manage the entity identifier within the database implementation. A *foreign key* column must have the same datatype as the column being referenced.
- The only notable data-type not shown in this db is a date/time format. They can be manipulated using the `<` and `>` symbols, making them very flexible. Additionally, *timestamps* and *datetime* fields can be updated with the current time automatically by the *RDBMS*.

Examining Tables:

The list of tables can be shown and an individual table described with these commands respectively *(not on specification)*:

```
show tables;
describe table_name;
```

11.6.2 Altering Tables

Once created, a table's definition may need to be changed. The `Alter Table` command is used for this.

Adding a constraint to a table is done like this (not on specification):

```
ALTER TABLE engine ADD FOREIGN KEY (supplier_id) REFERENCES supplier (id) ON DELETE
↪  CASCADE ON UPDATE CASCADE;
```

Adding a column is also possible:

```
ALTER TABLE engine_option ADD engine_quantity INT(1);
```

Existing columns can be renamed with ease:

```
ALTER TABLE engine_option RENAME COLUMN engine_quantity TO engine_number;
```

The definition of a column can be modified:

```
ALTER TABLE engine_option MODIFY COLUMN engine_number INT(255) NOT NULL;
```

Columns can also be dropped or deleted:

```
ALTER TABLE engine_option DROP COLUMN engine_number;
```

11.6.3 Inserting Data

The general syntax for a simple insertion is:

```
INSERT INTO table_name (first_column_name, second_column_name) VALUES
    (first_value, second_value),
    (third_value, fourth_value);
```

Here are the insert statements for the aircraft demo:

```
INSERT INTO supplier (id, name) VALUES
            ('PW', 'Pratt & Whitney'),
            ('CFM', 'CFM International'),
            ('RR', 'Rolls Royce'),
            ('EA', 'Engine Alliance');

INSERT INTO aircraft (id, name, price) VALUES
            (123, 'A320', 100000000),
            (243, '737', 100000000),
            (156, 'A380', 400000000),
            (457, 'Typhoon', 110000000);

INSERT INTO engine (id, name, supplier_id) VALUES
            ('1100G', 'geared turbofan', 'PW'),
            ('1A', 'high bypass engine', 'CFM'),
            ('1B', 'high bypass engine', 'CFM'),
            ('900', 'Trent 900', 'RR'),
            ('7000', 'EA GP7000', 'EA'),
            ('2000', 'EJ2000', 'RR');

INSERT INTO engine_option (aircraft_id, engine_id, engine_number) VALUES
            (123, '1100G', 2),
            (123, '1A', 2),
            (243, '1100G', 2),
            (243, '1B', 2),
            (156, '900', 4),
            (156, '7000', 4),
            (457, '2000', 2);
```

11.6.4 Selecting Data (One Table)

Once there is some data in the database, organised in *3nf*, it is possible to start making some queries to return records, using the `SELECT` statement.

The most simple query returns data from just one table. Here are some examples:

```
SELECT * FROM aircraft;
SELECT name FROM supplier;
SELECT name, supplier_id FROM engine;
```

The first command returns the values from all columns of the `aircraft` table, while the others specify certain, comma-separated columns to display.

11.6.5 Selecting Data (Multiple Tables)

There are a number of ways to select data from multiple tables, including *joins* and *sub-selects*, neither of which are on the a-level specification.

Here are some example queries across multiple tables:

```
SELECT supplier.name, engine.name FROM supplier, engine WHERE supplier.id =
↪  engine.supplier_id;

SELECT supplier.name, engine.name FROM supplier, engine WHERE supplier.id =
↪  engine.supplier_id AND supplier.id = 'CFM';
```

Note that the `WHERE` clause provides the condition for the selection. In the second query, `AND` is used to provide a second condition.

11.6.6 Ordering Results

The results of a select statement can be ordered in a certain way, using the `ORDER BY` statement. Here is an example:

```
SELECT * FROM aircraft ORDER BY id DESC;
```

This will return all the results in order from highest id to lowest id. If the sort column is a string, the results will be sorted alphabetically. Multiple Columns can be specified as sort column, for example results might be sorted alphabetically by country and then city.

11.6.7 Full Query

For a fully fledged database query, this is the default syntax:

```
SELECT first_column_name, second_column_name
FROM first_table_name, second_table_name
WHERE first_condition AND second_condition
ORDER BY column_name DESC/ASC;
```

11.6.8 Updating Records

In addition to the table's definition, records can be changed. The `UPDATE` command is used to do this. Here are a few examples:

```
UPDATE aircraft
SET name = 'a320'
WHERE id = 123;

UPDATE aircraft
SET price = price+20000000
WHERE id = 243;
```

Once again, the `WHERE` clause is used to supply the conditions for the statement.

11.7 Client-Server Databases

Many modern *RDBMS* support a client-server model, where the database is stored on a central server and many clients can connect across the network. This is useful in retail, for example, where individual stores can access the same information about products, without the need for the data to be stored locally. Other benefits include the consistency of data, which can be guaranteed by storing it in one place only. In addition, the integrity of the data is upheld, as backups and access rights are managed centrally.

11.7.1 Record Locking

If one client needs to modify a record, the encapsulating block of data is copied to the client workstation. Once the data has been modified, the block is submitted back to the central server.

If another client checked out the same block during this period, and resubmitted it after the first client had finished with the data, the second client may find that the block present in the server does not match the block it took out and modified. If the second client continued to submit its changes anyway, the update by the first client will be lost.

To solve this problem, *record locking* was introduced, whereby a block being modified by a client cannot be accessed until it is returned to the server. This prevents simultaneous access completely, upholding the consistency of the data.

11.7.2 Deadlock

The problem with *record locking* is *dead-lock*. It is possible that two different clients *(1 & 2)* have taken out different blocks of data *(A & B)* and now they both want to access the data held by the other client.

- Client 1 is holding *block A* and waiting for *block B*
- Client 2 is holding *block B* and waiting for *block A*

Neither client returns the data and both wait for one another unknowingly. There are a number of techniques used to solve *deadlock*.

11.7.3 Serialisation & Timestamp Ordering

In order to prevent concurrent access, each record has two timestamps, *read* and *write* which are set whenever a transaction is applied. When a transaction begins it is also given a timestamp.

Whenever transactions need to be applied, the timestamps are checked and the transaction with the first timestamp is applied first. The timestamp of a transaction is compared against the timestamps of the affected records.

Using all of this information, the DB management software can apply transactions in the order they began and keep the data consistent.

11.7.4 Commitment Ordering

Commitment Ordering is another *serialisation* technique, used to manage concurrent access. In addition to the time transactions were initialised, modifications are ordered by the dependencies on one another and the data stored in the DB.

11.8 Approach to Problem Solving

11.8.1 Analysis

Before development begins, the requirements of the client must be established. This includes identifying the shortcomings of existing solutions. Some of the factors to consider include the client's existing data and how it will be handled by the new system.

11.8.2 Agile Modelling

A large project is sometimes broken down into smaller parts like implementing a certain feature. During development, certain parts of the project may be developed at different rates.

The analysis of one feature may be concluded after the implementation of another. Developers may need to conduct a feasibility study, proving one requirement can be met, before continuing work on others. (feasibility study is no longer in specification)

Working with a client can be an *iterative process*. The client will provide feedback regularly, as prototypes are built. This allows the program to be *refined* as it is developed.

11.8.3 Design

After the requirements of the project have been finalised between the developers and the client, the developers can decide how the program will be made. Factors to consider include:

- input data
- data structures
- algorithms
- output data
- UI/UX
- security
- hardware requirements

11.8.4 Implementation

During the implementation section, the requirements identified in the analysis section are met using the techniques outlined in the design phase. Whilst the features are being implemented, it is important to keep to the *critical path*, the required features must be met before any others.

11.8.5 Testing

All inputs are tested with normal, boundary and erroneous data. Other tests include:

- Unit testing
- Module testing
- Sub-system testing
- System testing

Once the system is working, the client performs *acceptance testing*, making sure that the system works with their data and meets their requirements.

Testing is also an *iterative process*. Should the software fail the client's acceptance testing, for example, the developers may make some changes and run the new code through existing tests.

11.8.6 Evaluation

Three to six months after the delivery of the software, a post-implementation review may be conducted to determine the final quality and suitability of the system. The project may be judged on *effectiveness*, *usability* and *maintainability* by the client. Thw review is a good opportunity to discuss improvements to the software.

12 OOP and Functional Programming

12.1 Imperative Programming

Early high-level languages fell into the *imperative* category, meaning a program consisted of a series of steps, executed in order, to solve a problem. This paradigm developed naturally, as programs featured explicit steps for a computer to perform, in contrast to *declarative* languages.

12.2 Procedural Languages

Programs that divided these steps into separate functions and subroutines were called *procedural* languages. The control flow of a procedural program follows a sequence of subroutine calls. Data in such programs is held separately in primitive types. Each piece of data exists in a certain scope, whether that is *global* or *local*, and data can be passed to a sub-routine as an argument.

Sub-routines can be divided into two types:

- *procedures*, which may have parameters and return values, but also cause side-effects
- *functions*, which may have parameters and must have return values, while causing no side-effects

Many modern imperative languages support both types of sub-routine. *Purely functional* languages are those which only support the latter, the benefits of which are explained in the functional programming section.

12.3 Object Orientation

More modern programming languages associated data with behaviour, creating *objects*. In an object orientated program, both real world data and program code are considered objects.

Here are some notes on how to use some of python's OOP features: `https://github.com/alexander-neville/docs/blob/main/python.org`

12.3.1 Implementations of OOP

Since *OOP* is a paradigm, many languages support similar, although slightly different, programming techniques for working with types and objects. Generally speaking a named instance of a object is a *reference type*, holding a pointer to where the object itself resides in memory.

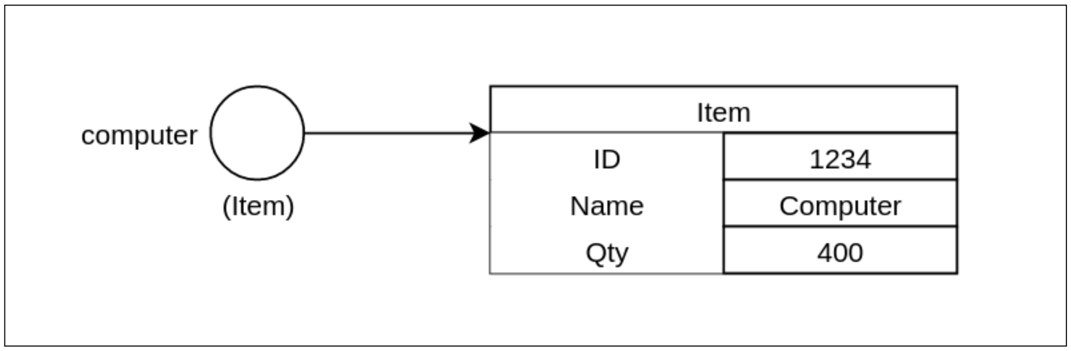

Figure 9: Reference type

There are many different implementations of objects across different languages, from *structs* in C and *object literals* in Javascript to *classes* in python and C++.

12.3.2 Classes

While it is possible to work in the object orientated style without classes, it is a useful construct pervasive in most OOP languages. A class is a blueprint for an object, defining the data and behaviour of an object.

Within the class for an object, certain attributes may be marked as *private*, preventing the state being modified outside of the *instantiated* object's own behaviours. So called *information hiding* is an important improvement over traditional *procedural* code, as it reduces *mutability* and unexpected *side-effects*.

This is an example of a class, written in pseudocode:

```
item = Class
    Public
        Function GetStockLevel
        Procedure UpdateStockLevel
        Procedure SellItem
        Procedure Describe
    Private
        Id: Integer
        StockLevel: Integer
        Name: String
        Description: String
End
```

Most attributes of the object are marked as private, while the object's behaviour is usually public. This is how other parts of the code interact with the object. Sometimes *getter* and *setter* methods are used to manipulate the attributes of a method. In this class, `GetStockLevel()` and `UpdateStockLevel()` perform this role.

12.3.3 Encapsulation

The association of data with behaviour, especially the use of *getter* and *setter* sub-routines, is called *encapsulation*. By including the methods which operate on the data with the data itself, in a single object, access to the data from the program at large is restricted, hiding implantation details and better maintaining state across the program. Additionally, an interface can easily be built around the data, by modelling objects with classes, which makes the organisation of a larger project much simpler.

12.3.4 Instantiation

A class may also include a *constructor*, a sub-routine used to create an *instance* of and object. This sub-routine is run when an object is *instantiated* from a class. Each instantiated object has all the attributes and methods defined by the class. The precise value of each of these properties is associated with the instance, so it is possible to have many different instances of a single object. In most programming languages an instance is a reference type.

12.3.5 Polymorphism

Using a combination of object orientated techniques, a single *interface* can be shared by a number of different objects (instantiated from different classes). This means that a single *message* applied to a number of instances will cause a different action depending on the type of object receiving the message. For this technique to work, each object must implement some behaviour for the *message*.

A *message* is simply a call to an object's methods. For example:

```
object1.test_message(inputs)
object2.test_message(inputs)
```

If the program is *polymorphic*, the same method (in this case '`test_message`') can be called on both `object1` and `object2`, even if they are instantiated from different classes. The results will depend on the objects implementation of that *message*. Note that the classes for these objects must define this behaviour. Therefore, the name of the methods and the call pattern are identical, but the contents of the method could be very different

There are many constructs, across programming languages, that are used for building an interface over many classes. One example is *inheritance*, which is a object orientated technique in itself, while another is the use of an *interface*.

12.3.6 Interfaces

A programming interface stipulates the messages which a related class must respond to. The interface does not define any behaviour itself and the behaviour of each subclass for the messages does not need to be known.

This is how an interface would look in pseudocode:

```
Public interface BankAccount
    Procedure GetAccNum
    Procedure GetSortCode
    Function Withdraw (Amount)
    Function CalcInterest (TimePeriod)
End
```

The interface might be implemented like this (pseudocode example):

```
Class ReputableFirm implements BankAccount
    Public
        Procedure GetAccNum
        Procedure GetSortCode
        Function Withdraw (Amount)
        Function CalcInterest (TimePeriod)
    Private
        AccNum: Integer
        SortCode: Integer
        InterestRate: Float
        Balance: Double
End

LoanShark = Class implementing BankAccount
    Public
        Procedure GetAccNum
        Procedure GetSortCode
        Function Withdraw (Amount)
        Function CalcInterest (TimePeriod)
    Private
        AccNum: Integer
        SortCode: Integer
        InterestRate: Double
        Balance: Float
End
```

Note that both classes implement the same public functions (same definition and parameters), however some of the attributes of either class differ. For example, an account with a loan shark may have an extortionate amount of interest, hence the rate is stored as a `Double` rather than a `Float`. The subclasses are responsible for implementing the methods specified by the interface, however the details of that implementation are unique to the classes themselves.

12.3.7 Inheritance

A plain *interface* is more suitable for vaguely unrelated objects, whereas *inheritance* is designed for objects with behaviour in common. If inheritance is a good option, it can be said that the objects have an *"is a"* relationship. For example a cat *is an* animal.

An object may *inherit* from multiple other classes at once, or inherit from a single class which also inherited from another class in turn. A subclass has all the properties and methods of the superclass/parent class. With each level of inheritance, a subclass can do any of these things:

- add another method to those inherited from the parent class(es)
- *override* or change the behaviour of a certain inherited method entirely
- modify inherited behaviour (make a call to parent's implementation of a method)

In some programming languages it is possible to define entirely abstract classes, which *must* be inherited from (an abstract class cannot be instantiated on its own). This differs from an *interface* because the abstract class does implement some base behaviour.

Here is the example of inheritance from the textbook (pg 354):

```
Animal = Class
    Public
        Procedure MoveLeft
        Procedure MoveRight
    Protected
        Position: Integer
End

Cat = SubClass (Animal)
    Public
        Procedure MoveLeft (override)
        Procedure EatMouse
    Private
        Name: String
End
```

Each subclass which inherits from the `Animal` parent class will have all the methods of the parent class. In the `Cat` subclass, the `MoveLeft` method is *overriden*, but the `MoveRight` method is not changed. This means that a cat object will move right in the same way as a generic animal if that message is sent to the object. The `Cat` object also adds the `EatMouse` method, which is specific to a cat. Here is a diagram of inheritance:

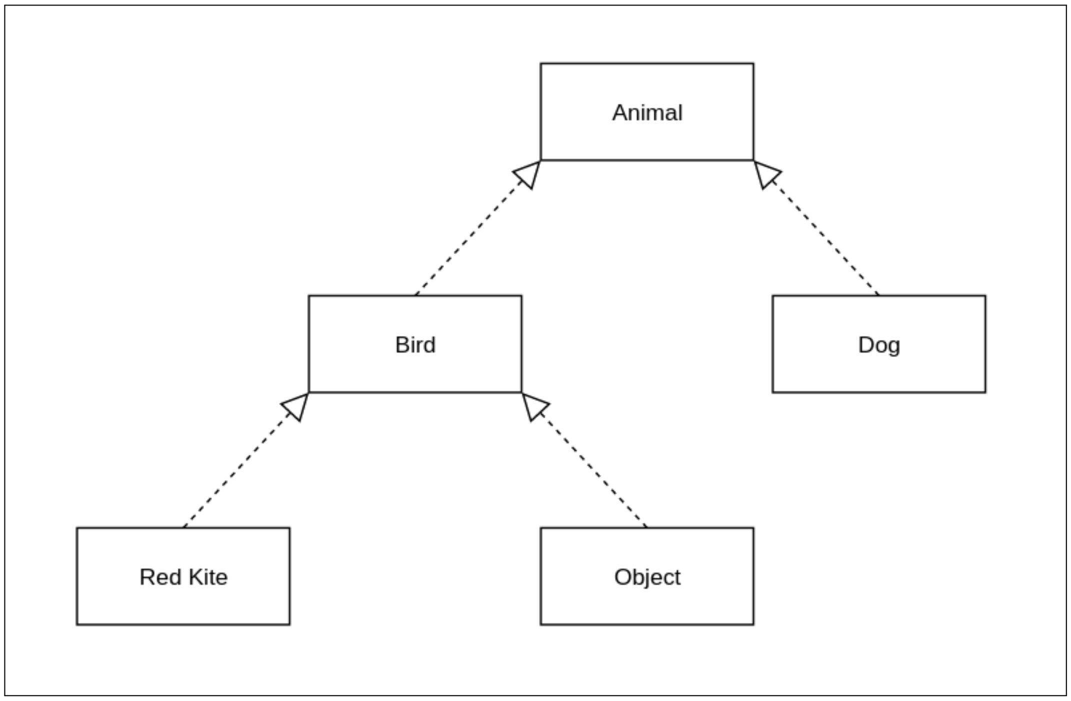

Figure 10: Diagram of multiple inheritance

12.3.8 Demonstration

This is an example of inheritance and polymorphism written in python, but the theory is applicable to the a-level topic.

The first part of the program is the definition of the base class, in this case it is a model of an animal. This is not marked as abstract, but its role is parent class and its purpose is defining a standard interface and some default behaviour. For the sake of simplicity, the base class (and derived classes) only have one method in this example.

```
class Animal(object):


    def __init__(self):
        pass


    def describe(self):
        print("i am a generic animal")
```

With the base class established, two subclasses are defined, inheriting from the `Animal` class. These two classes model a bird and mammal respectively. They have more specific properties than the parent class and the `describe` method is overriden. The `Bird` class includes a call to the superclass' implementation of the message, as well as adding its own behaviour. In addition, the bird class has an additional property.

```
class Bird(Animal):

    def __init__(self):
        self.can_fly = True

    def describe(self):
        Animal.describe(self)
        print("i am a bird\n")


class Mammal(Animal):

    def __init__(self):
        pass

    def describe(self):
        print("i am a mammal\n")
```

A sub-routine is declared which can test the polymorphic design of the program. An instance object is passed to the sub-routine and the `describe()` method is called on the reference variable, irrespective of the object passed in. The outcome of the sub-routine will depend on the type of the object passed in, rather than the code in the sub-routine.

```
def test_object(animal):
    animal.describe()
```

The last part of the program is simply some driver code; two objects are instantiated and passed to the `test_object()` sub-routine.

```
bird = Bird()
mammal = Mammal()

test_object(bird)
test_object(mammal)
```

The output succinctly demonstrates the principle of polymorphism. Both objects share a common interface, which can be called upon by an external unit of code. The resulting output is a consequence of the object's type, not the operation requested.

```
i am a generic animal
i am a bird

i am a mammal
```

12.3.9 Association

When two objects have a relationship that is not an *"is a"* relationship, association by *composition* or *aggregation* may be more suitable. Such a relationship is usually a *"has a"* relationship, for example a house *has a* kitchen. There is usually some ownership involved, otherwise the objects can exist independently. The difference between the two types of association lies in the *life cycle dependency* of the contained classes.

- during *aggregation* association, the subclasses continue to exist without the container class, eg. people in a sports team or a company still exist if either is disbanded
- during *composition* association, the contained classes are destroyed along with the container class, eg. rooms in a building disappear if the building is knocked down

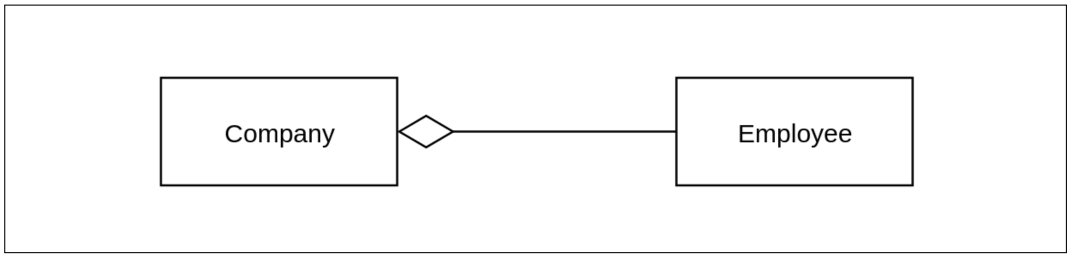

Figure 11: Association by aggregation

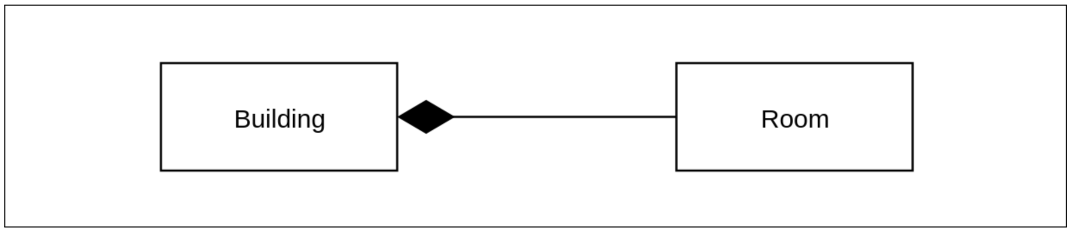

Figure 12: Associaiton by composition

12.3.10 Access Modifiers

In programming languages, *information hiding* can be enforced, so that program code must use an object's interface to retrieve data. There are generally three main modifiers:

- *public* properties/methods can be used from anywhere in a codebase
- *private* properties/methods can only be accessed or run from within an object's own behaviours
- *protected* properties lie somewhere in between, depending on the language.

12.3.11 Class Diagrams

As is the case with DB design, there are a number of different diagrams designed to make working in the OOP style easier to plan. One common standard is the UML *(unified modelling language)* diagram.

A single `-` sign is a private attribute, a `+` is a public attribute/method and `#` is used for protected attributes.

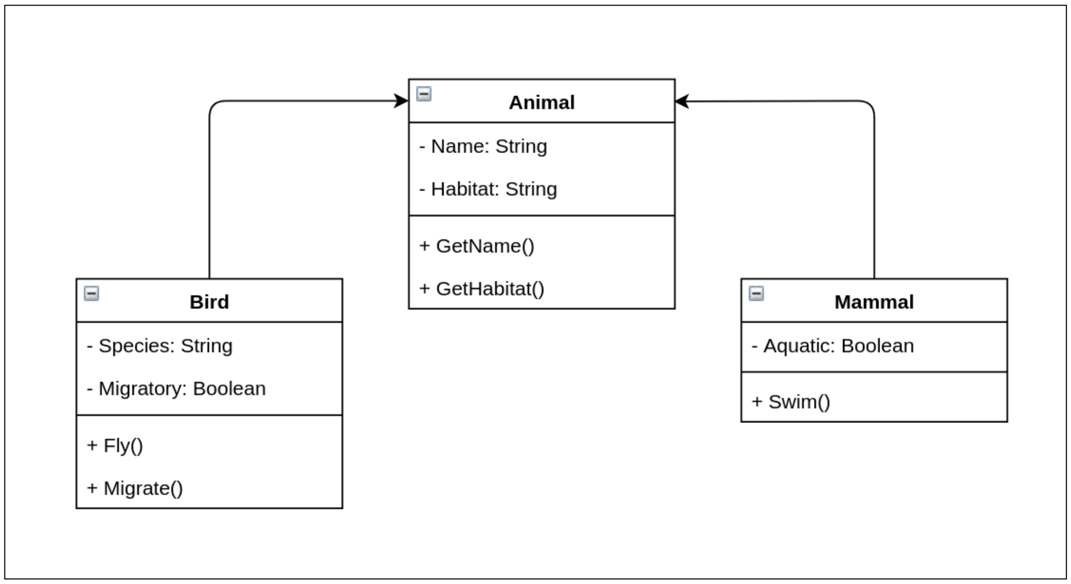

Figure 13: A UML diagram for the standard animal example

12.3.12 Advantages of OOP

OOP is preferred to earlier procedural techniques for a number of reasons:

- planning is more important/worthwhile in the OOP paradigm
- encapsulation means the implementation of data needs only happen once
- an old OOP codebase is easily modifiable, by adjusting classes
- general modularity and ease of debugging/maintenance

12.4 Functional Programming

Another significant programming paradigm is the functional technique, in which composing functions are used to map values to one another, as opposed to the execution of a sequence of imperative statements. Functions are *first-class citizens* in functional languages; they can be bound, passed and returned from other functions and can generally be used like any other data type.

Functional programming is derived from *lambda calculus*, a formal system of computation which uses only functions. Functional programs, particularly *pure* functional programs, can be formally verified as correct. Larger programs can be constructed, or *composed*, from these smaller modular parts, in the knowledge that each function behaves as expected. This paradigm makes development easier and faster, without the risk of side-effects causing issues or bugs in the program.

12.4.1 First-class and Higher-order Functions

A *higher-order* function is one which can either take functions as arguments or return them as results. Higher-order functions describe functions which operate on other functions, which implies the use of first-class functions, a computer science term for programming entities with no restriction on their use.

The examples given below demonstrate the common higher-order functions `foldl`, `map` and `filter`. The `fold` function recursively collapses a data structure into a single value according to the given function, the `map` function applies the specified function to each element in a list and the `filter` function returns a list of elements from the original list satisfying a particular condition.

```
Prelude> map (+ 5) [1,2,3,4]
[6,7,8,9]
Prelude> filter (< 5) [3,4,5,6]
[3,4]
Prelude> foldl (+) 0 [1,2,3,4]
10
```

12.4.2 Purely Functional Languages

A *pure* function is one which causes no side-effects and does not edit a global state. This includes any side-effects in memory or on disk via I/O.

- If a function is pure and is called with a set of arguments of the same specification, the same output can be expected if the same call is made repeatedly. This is called *referential transparency* or *idempotence*.

- If there is no common dependency between two pure functions, the can be executed in any order, or at the same time. This makes purely functional code *thread-safe* and *lazy*; functions can be applied in any order as required.
- If the result of a computation is not used and not required elsewhere, the computation or evaluation of a particular function need not occur at all. This gives compilers greater freedom to optimise code.

12.4.3 Recursion

Iteration, which is common in procedural languages, is achieved with recursion in functional languages. A recursive function invokes itself until a base case is reached. Recursion does require that a stack is maintained to handle the scopes, which can become expensive in space. Certain compilers can recognise types of recursion and provide optimisation.

12.4.4 Function Application

In purely functional languages such as Haskell, functions may only have a single argument. Functions with more arguments use *partial application*, not to be confused with *currying* to break a function with n arguments into n functions with 1 argument.

Consider the below example. The addition operator is bound to `a`, and the resulting function can be used as expected. If `a` is called with only a single argument, an error is produced, indicating that the returned value is a function which cannot be printed. If instead the result is bound to another identifier, the return value is proven to be a function itself, operating on a single argument.

```
Prelude> a = (+)
Prelude> a 2 5
7
Prelude> a 2

<interactive>:8:1: error:
    • No instance for (Show (Integer -> Integer))
        arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
Prelude> b = a 2
Prelude> b 5
7
Prelude>
```

12.5 Big Data

Big Data is a term used to describe datasets which are very large in volume, which either exceed the memory capacity of computers used to process the data or exceed the storage capacity of a single device or both. Such data is often *unstructured*, it is impossible to store in a relational database.

Functional programming lends itself towards big data. The lack of side effects and ease of parallelism makes it easier to deploy functional algorithms across many cores and computers to process data efficiently. Other features of the functional paradigm, such as immutability make development and debugging of algorithms easier, especially in complicated and long running programs.

An alternative way of organising data to typical relational databases is the fact-based model. Immutable facts are recorded once, with a timestamp and are never modified or deleted. If the fact is to be changed, a second fact is added with a more recent timestamp.