Some of the Big Ideas in Computer Science

Joseph Cunningham

Contents

Clean architecture
Gang of four
TODO: unity of terminology!

Part I

Background

Chapter 1

What is a computer?

What is data? Also binary Turing machine: undecidability, the halting problem, Lambda calculus Abacus Von Neumann / computer architecture (+infra) Church-turing thesis (+ materialism?) Minimal working computer

Chapter 2

Organization of computer systems

A computer system can be divided roughly into four components:

1. The hardware;
2. The operating system;
3. The application programs;
4. The users.

2.1 Hardware

2.1.1 The CPU

assembly / machine language
address of execution / program counter

2.1.2 Storage

Pointers

2.1.2.1 Design parameters

- Speed (read / write)
- Size
- Volatility
- Ability to be written to (yes - RW / no / limited - WORM)

2.1.2.2 Types of storage

ROM Also EEPROM. Firmware.

Registers

Cache

Main memory (RAM)

Secondary memory

External (tertiary) storage

2.1.3 I/O devices and controllers

2.2 Computer-system architecture

TODO: what is it? Configuration of pieces of hardware.

2.2.1 System design

2.2.1.1 Computing environments

Traditional computing

Mobile computing Includes GPS, accelerometers etc.

Distributed systems Networks characterised based on distance between nodes:

- Personal-area network (PAN)
- Local-area network (LAN)
- Metropolitan-area network (MAN)
- Wide-area network (WAN)

(Mostly LAN - WAN)

- Client-server computing
- Peer-to-peer. Needs discovery protocol.

Cloud computing Public or private: for anyone willing to pay or company internal.

- Software as a service (SaaS): one or more applications available via the internet.
- Platform as a service (PaaS): a software stack ready for application use via the internet.
- Infrastructure as a service (IaaS): servoers or storage available over the internet.

Real-time embedded systems

2.2.1.2 Building systems

Interrupts from hardware or software (system call = monitor call)
Interrupts must be handled quickly. Generally, a table of pointers to the interrupt service routines is stored in low memory. This is called the interrupt vector and is indexed by a unique device number, given with the interrupt request.
The address of the interrupted instruction must also be saved. Older systems simply saved this at a fixed location. More modern architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state (for example by modifying register values) it must explicitly save the current state.
After the interrupt is serviced, the saved return address is loaded into the program counter and the interrupted computation resumes.

Firmware Bootstrap program to get computer started.

2.2.1.3 Some architectures

Single-processor systems + use of special-purpose microprocessors (in disk / keyboard / ...)

Multiprocessing Multipocessor systems, also known as parallel systems or tightly coupled systems, have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
Multiprocessor systems have three main advantages:

1. **Increased throughput**. More processors means more computational power. The computational power does not scale linearly with the number of processors, though. There is some overhead due to process coordination. Also competition for resources such as memory can occur.

2. **Economy of scale**. A multiprocessor system is typically cheaper than multiple single-processor systems, because peripherals can be shared.

3. **Increased reliability**. Failure of a single processor does not need to halt the whole operation.

    - The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation.
    - If a system has enough redundancy that operation is unaffected by failure, it is called fault tolerant.

There are two main types of multi-processor systems:

1. In systems using asymmetric multiprocessing there is a master CPU that coordinates the slave CPUs.

2. In symmetric multiprocessing (SMP) there is no such hierarchical relationship. All CPUs are equals.

Multiprocessor systems may have *uniform memory access (UMA)* or *non-uniform memory access (NUMA)*. With UMA accessing any RAM from any CPU takes the same amount of time.

Multiprocessing may also be achieved by including multiple computing **cores** on a single chip. This has the advantage of fast on-chip communication and a minimal increase in power requirements.
Blade servers multiple processor, I/O and networking boards in the same chassis.

Clustered systems Clustered systems consist of several individual systems, or nodes, joined together. The definition of the term *clustered* is not concrete.
Clustering is often used to provide *high-availability* or *high-performance*.
Applications must be specifically written to take advantage of the cluster. This is called parallelisation.
Clusters may **symmetric** or **asymmetric**. In asymmetric clustering, one machine is in hot-standby mode, ready to jump in if the active server fails.

Example

Beowulf clusters are clusters built using commodity hardware using a set of open source packages for high-performance computing tasks.

2.2.1.4 Direct memory access (DMA)

2.2.2 Instruction set architecture (ISA)

RISC, CISC

2.2.3 Microarchitecture

Von Neumann?

2.3 Operating systems

An operating system is a program that manages the computer hardware. It acts as an intermediary between the user and the hardware. The purpose of of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

An operating system may be thought of as a control program and as a *resource allocator*.
There are many different types of computer systems with different design goals. For this reason we have no completely adequate definition of what an operating system really is or exactly which programs are part of the operating system and which are not. A simple approach is to call everything a vendor ships the "operating system".
The software a vendor ships can be split into three categories:

1. The kernel which is the one program running at all times on the computer;
2. Systems programs, which are associated with the operating system, but are not part of the kernel;
3. Application programs, which include all programs not associated with the operation of the system.

Often people use the term "operating system" to refer specifically to the kernel.
ALSO: Microsoft case for bundling IE.
Now: middleware, especially in mobile OSs: software that provides services to software applications beyond those available from the operating system, such as for databases, multimedia and graphics.
Operating systems typically provide a set of services that are helpful to the user:

- A user interface (UI). This may be graphical (GUI) or text based.
- Program execution.
- I/O operations
- File-system manipulation
- Communication between processes. E.g. via shared memory or message passing.
- Error detection. For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Another set of functions ensures the efficient operation of the system:

- Resource allocation.
- Accounting, i.e. keeping track of which users use which computing resources.
- Protection and security.

Some relevant design parameters for operating systems include

- Ease of use;
- Performance;
- Fair resource allocation;
- Maximisation of resource utilisation;
- Real-time responsiveness;
- Battery life.

2.3.1 Interfacing with the system

2.3.1.1 Command interpreter

Some operating systems include the command interpreter in the kernel, but many, like Windows and UNIX, treat the command interpreter as a special program that is running when a user first logs on. On systems with multiple command interpreters to choose from, the interpreters are known as shells.
The user operates the command interpreter by typing in commands. Some command interpreters then jump to code execute the command issued, using the appropriate system calls. An alternative approach (used by UNIX among others) is to let the interpreter call a relevant system program. This way new commands can very easily be added to the system.

2.3.1.2 Graphical user interfaces

TODO skeuomorph

WIMP and post-WIMP interfaces

Window managers

- Compositing window managers
- Stacking window managers
- Tiling window managers
- Dynamic window managers

Desktop environments

Mobile interfaces gestures

2.3.2 CPU allocation

2.3.2.1 Multiprogramming and time sharing

In general a single program cannot keep the CPU at all times. Also a single user generally is running multiple programs simultaneously. The solution is multiprogramming: the system keeps several *jobs* in memory at all times. The operating system picks a job to execute for a while until it needs to wait for some task (such as an I/O operation) to complete. At that point the operating system switches to a different job from the job pool.
A program loaded into memory and executing is called a process.
Time sharing or multitasking operating systems switch jobs often enough that the user can interact with each program while it is running.
A time-shared operating system allows many users to share the computer simultaneously.

2.3.2.2 Timer

The operating system must keep control of the CPU in order to be able to provide multitasking. A user program must always give control of the CPU back to the operating system and may not enter an infinite loop or just fail and not return control to the operating system.
To ensure proper operation, a timer is set which sends an interrupt to the CPU to hand control back to the operating system. Before giving control to a user program, the operating system makes sure to set the timer.

2.3.3 Dual-mode operation

Clearly the operating system needs to be able to perform operations that the user's programs may never, such as setting the timer or modifying the job pool and scheduling information.
The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution. A bit, called the mode bit is added to indicate the current mode: user mode or kernel mode (also called supervisor mode, system mode or privileged mode).

Some of the machine instructions are privileged instructions that may only be executed in kernel mode. Examples include

- I/O control;
- time management;
- interrupt management.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
I/O must be reserved for the operating system because otherwise a user program may try to overwrite the operating system or other programs or two programs may try writing to a device at the same time.

2.3.4 System calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

2.3.5 System programs

system processes or system deamons: provide services outide the kernel. run entire time kernel is running?

2.3.6 Storage management

One of the major ways operating systems provide resource management is by abstracting the physical properties of its storage devices to define a logical storage unit, the file.
A file is a collection of related information defined by its creator. Files are used to represent both data and programs. Data files may be binary or text-based.
The operating system controls who can access which files in what way.
Caching. Cache management. Cache coherency. TODO OSv9 p28

Chapter 3

Abstraction and mathematics

Part II

Theoretical computer science

Chapter 1

Data structures

Numbers: Endianness, 2's complement, Floats Text: Ascii example (full encoding discussion later?), Char / string Arrays Hash Linked lists ... Language specific examples (eg deque, vector, ...)

1.1 Information theory

1.2 Elementary data structures

1.3 Linear lists

1.3.1 Stacks, queues and deques

1.3.2 Sequential allocation

1.3.3 Linked allocation

1.3.4 Circular lists

1.3.5 Doubly linked lists

1.3.6 Arrays and orthogonal lists

1.4 Hash tables

?

1.5 Graphs and trees

1.5.1 Mathematical properties

1.5.2 Binary search trees

1.5.3 Red-black trees

1.5.4 Augmenting data structures

1.5.5 B-trees

1.5.6 Fibonacci heaps

1.5.7 Van Emde Boas trees

1.6 Multilinked structures

1.7 Dynamic storage allocation

1.7.1 Data structures for disjoint sets

Chapter 2

Computation theory

2.1 Models of computation

Stack machine (0-operand machine) Accumulator machine (1-operand machine) Register machine (2,3,... operand machine)
Expressive power
http://cs.brown.edu/people/jsavage/book/pdfs/ModelsOfComputation.pdf

2.1.1 Sequential models

2.1.1.1 Finite state machines

2.1.1.2 Pushdown automata

2.1.1.3 Turing machine

2.1.2 Register machines

2.1.2.1 Counter machine

2.1.2.2 Pointer machine

2.1.2.3 Random-access machine (RAM)

2.1.2.4 Random-access stored-program machine model (RASP)

2.1.3 Functional models

2.1.3.1 Lambda calculus

2.1.3.2 Recursive calculus

2.1.3.3 Combinatory logic

2.1.3.4 Abstract rewriting systems

2.1.4 Concurrent models

2.1.4.1 Cellular automata

2.1.4.2 Kahn process networks

2.1.4.3 Petri nets

2.1.4.4 Synchronous data flow

2.1.4.5 Interaction nets

2.1.4.6 Actor model

2.2 Computability theory

2.3 Computational complexity

Depends on model of computation. TODO Cell-probe model

Chapter 3

Program verification

loop invariants?

3.1 Axiomatic

Hoare, 1969

3.2 Weakest precondition

Dijkstra, 1972

3.3 Algebraic data types

Guttag, 1975

3.4 Functional correctness

Mills, 1975

Chapter 4

Algorithms

Time vs space complexity Tradeoff Proofs of correctness Sort Search Pattern matching

4.1 Foundations

4.2 Arithmetic

4.3 Sorting and order statistics

4.3.1 Insertion sort

4.3.2 Heapsort

4.3.3 Quicksort

4.3.4 Sorting in linear time

4.3.5 Medians and order statistics

4.4 Graph (searching) algorithms

4.4.1 Elementary algorithms

4.4.1.1 Breadth-first search

4.4.1.2 Depth-first search

4.4.1.3 Topological sort

4.4.2 Minimum spanning trees

4.4.3 Single-source shortest path

4.4.4 All-pairs shortest path

4.4.5 Maximum flow

4.5 Random numbers

4.5.1 Generating uniform random numbers

4.5.2 Statistical tests

4.5.3 Random sampling and shuffling

4.5.3.1 Fisher-Yates shuffle

4.6 Multitheaded algorithms

4.7 Matrix operations

4.8 Linear programming

4.9 Polynomials and the FFT

4.10 Number-theoretic algorithms

4.11 String matching

4.12 Computational geometry

4.13 NP-completeness

Chapter 5

Data types and type theory

5.1 Introduction: what and why?

In computer science types provide a lightweight formal method for theoretical analysis of some aspects of computer programs.
They are related to the broader mathematical field of type theory. Type theory was originally conceived as a way to avoid logical paradoxes, like Russell's paradox.
The name "type system" can refer to a large number of different systems, making it difficult to give a general definition. (TODO cite Pierce) suggests the following

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

A type system can be seen as a kind of static approximation of the runtime behaviour of the program. Sometimes the moniker "static" is explicitly added, to distinguish it from dynamic typing, which is arguably a misnomer (and should maybe be referred to as dynamic checking, but the usage is standard).
Most type systems can only guarantee that well-typed programs are free from certain types of errors. However in principle it should be possible to make type annotations encode an arbitrary specification. In this case the type checker would become a *proof* checker.
Type checking is / should be automatic. Here we are interested in practical applications, so we are most interested in methods that are not just automatable in principle, but actually have known, efficient algorithms.

5.1.1 What type systems are good for

1. **Detecting errors**. Not only trivial mental slips, but also deeper conceptual errors.
2. **Abstraction**. Interfaces can be seen the the "type of the module".
3. **Documentation**. Types help make clear what variables and parameters are supposed to do. This documentation is built into the program and thus cannot go out of date.
4. **Language safety**. This concept is not well defined, but we can say that a safe language *protects its own abstractions*. In particular it means programmers can not (unwittingly) use data that was supposed to be a string as a number or write past the end of an array.

In extreme cases a program may damage the data structures of the runtime system and itself. One of the ways to achieve language safety is using is having a good type system, although often some sort of dynamic checking (e.g. for array bounds) is also needed.

There are also some other ways to view language safety:

- One may view a safe language as one that is completely defined by its manual. A language like C is not safe because,e.g., pointer arithmetic depends on the details of memory management. This is compiler specific and not part of the specification.
- Another perspective on language safety is given by defining trapped and untrapped run-time errors. Trapped errors cause execution to stop immediately, while computation can continue with untrapped errors.

  A safe language is then one that prevents untrapped errors at run time.

5. **Efficiency**. By giving the compiler more information, it can optimise more easily. Also static type checking is generally much faster than dynamic checking for guaranteeing language safety.

5.2 Untyped systems

5.3 Simple types

5.4 Subtyping

5.5 Recursive types

5.6 Polymorphism

5.7 Higher-order systems

Chapter 6

Database theory

Part III

Programming

Chapter 1

Language theory

regexes
CFG / CFL
Chomsky hierarchy
Lexers, parsers, translators and compilers

Chapter 2

Language design

Section consists mainly of bullet-points

2.1 General considerations

2.1.1 Desirable attributes

- **Suitability for application**.
- **Cost of use**. There are many relevant costs to take into account. They include
  - cost of program execution
  - cost of compilation
  - cost of program creation and testing
  - cost of maintenance
- **Programming environment**. Existence of external tools to ease development.
- **Portability**.
- **Readability** and, usually to a lesser extent, **writability**. Much more time is spent reading code than writing code, so optimise for readability, not terseness or compactness.
- **Conceptual clarity, simplicity and unity**. Different concepts and constructs should fit together understandably. This is called conceptual integrity.

  This also means there should be no ambiguity.
- **Promoting understandable programs**. This point is related to the last two. Ideally the language should promote the use of conceptually clear solutions.
- **Orthogonality**. As many different attributes of the language should fit together as possible. Downsides of orthogonality include allowing inefficient constructs and more difficult program verification.
- **Support for abstraction**.
- **Ease of program verification**. Techniques for program verification include

  - formal verification methods (proofs of correctness)
  - desk checking (i.e. visual inspection)
  - testing.

- **Context free grammar**.

2.1.2 Operating environments

Programming languages can be designed to work in different operating environments. The target environment has profound implications for the design of the language.

- Interactive environments (usually with a graphical user interface).
- Batch-processing environments where input (usually in the form of files) is given and the program computes the output.
- Embedded system environments.

2.1.3 Thoughts on optimizing information transmission

- It is easier to master static relations, than visualize processes evolving in time. We should shorten the conceptual gap between the static program (in text space) and the dynamic process (spread out in time). (Dijkstra)

2.2 Language paradigms

Language paradigms refer to the conceptual ideas that dictate the language's feature set. There are many different (mostly not mutually exclusive) programming paradigms. A language can support multiple paradigms.
The biggest distinction is between imperative and declarative languages.

- Imperative languages tell the computer *how* to perform calculations. Programs in imperative languages are typically a list of statements. Machine languages are imperative, so all computer languages eventually get translated into an imperative form. An imperative language may support the following paradigms:
  - **Procedural**, which groups instructions into procedures (also called subroutines).
  - **Object-oriented**, which groups instructions together with the data they operate on.
- Declarative languages declare the properties of the desired result but not how to compute it. Declarative paradigms include:
  - **Functional**, in which the desired result is declared as the value of a series of function applications.
  - **Logic programming**, in which the desired result is inferred from logical statements.

Several paradigms will be discussed in more depth when discussing concrete languages.

2.3 Some common language features

These are some common features shared by many languages. The information here is intentionally quite vague, as every language implements these features in different ways. The ideas presented here will be made much more concrete when looking at specific languages. Any particular language may only support some of these features.

- **Statements** are the basic building blocks of imperative languages. They instruct the computer which action to carry out next.
- **Data types** define how raw binary data should be interpreted. Examples include:
  - **Integers** are (positive or negative) whole numbers. Many programming languages can only represent numbers up to a certain size.
  - **Floats** are floating point numbers, i.e. real numbers. Many programming languages can only represent numbers up to a certain size and precision.
  - **Arrays** are lists of values.
  - **Pointers** refer to specific locations in memory.
- **Literals** are concrete representations of data that the computer can readily convert to a binary representation. Examples include numeric literals (such as 42 or 22.45), boolean literals (such as *true* and *false*), string literals (such as "Hello world") and array literals (such as $[1, 2, 5, 6]$).
- **Expressions** generally produce a value of a certain type. They may be literals, or a description of which operations to carry out in order to get the required value.
- **Variables** are named locations we can use to store values (usually of a particular type).
- **Control structures** control which statements get executed in which order.
  - **Conditionals** evaluate an expression. Depending on the result, the program does different things. The two most common constructs are
    - if-then-else statements (which execute the then part only if the expression evaluates to true) and
    - case-and-switch statements (which evaluate an expression and choose the correct course of action by comparing the value to several different case statements).
  - **Loops** execute a block of code multiple times.
    - While loops checks a condition every loop and while that condition is true it keeps on looping.
    - For loops generally do a loop for each value in a given set.
    - A language may also provide break and continue statements in order to, respectively, immediately break out of a loop and move on to the next iteration of the loop.
  - **Subroutines** are sequences of program instructions that each perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed. Often subroutines require arguments to operate on. A subroutine may also be called a **procedure**, a **function**, a **routine** or a **subprogram**, depending on the language. If the subroutine is bundled with data (such as in the object-oriented paradigm), it is conventionally called a **method**.

  - **Coroutines** are functions that can yield control to each other.
  - **Generators**, also known as *semicoroutines*, are functions that are in fact iterators, i.e. at each call it returns the next value in a sequence.
  - The **goto** statement causes the next statement to be executed to be the one at a particular **label**. Goto statements have been considered harmful by many programmers. The main points of the discussion are summarized in a later section.
  - **Exceptions** are *thrown* during anomalous or exceptional conditions. Hopefully there is then code that can *catch* the exception and handle accordingly. The exact notion of exceptions differs between languages. In some it is supposed to be used solely to handle abnormal, unpredictable or erroneous situations. In some it may be used for flow control structures.

- Some features typical for object oriented programming include:
  - **Objects** are usually containers with named fields containing data of different types as well as subroutines acting on the object (called methods).
  - **Classes** can specify what type of data objects contain as well as which methods they have. They can be seen as a way to define custom types.
  - New classes can be built based on old classes. This is called **inheritance**.
- **Regular expressions** are used to match (parts of) text, as discussed above.

2.4 Syntactic elements of a language

- **Character set**. Which encoding is to be used?
- **Identifiers** are names for things like variables and subroutines. Typically they are subject to some restrictions, like not being able to start with a number, or not being a reserved words.
- **Keywords** and **reserved words** are words that may not be used as identifiers, typically because they have some special meaning in the language.
- **Operator symbols**
- **Comments** consist of text in the program that is ignored during execution. The purpose of comments is to explain what the program is doing.
- **Blanks** are usually used to separate other syntactic elements. Sometimes they have other uses, such as newlines terminating statements in Python.
- **Delimiters** and **brackets**.

Here we may also remark upon the benefits of the (judicious) use of syntactic sugar. These are syntactic constructs that do not add any real functionality to the language, but merely provide an easier syntax for commonly used constructs.
This term has spawned some related concepts:

- Syntactic salt refers to a language feature designed to make it harder to write bad or incorrect code.
- Syntactic saccharin or syntactic syrup refers to gratuitous syntax that does not make programming any easier.

2.5 Practical aspects of data types

This is a more practical overview of some aspects of data types .

2.5.1 Data type specification

The basic elements of a specification of a data type are:

1. The possible **values** that data objects of that type may have.
2. The **attributes** that distinguish data objects of that type. Some of the attributes may be stored in a descriptor (also called a dope vector), as part of the data object during program execution. Examples may include name, data type, length etc.
3. The **operations** that define the possible manipulations of data objects of that type. We also generally wish there to be some sort of encapsulation so that the user of the type can only use the data in a correct way. This means that the user does not need to know the underlying implementation and is not permitted to directly manipulate the hidden implementation details.

2.5.2 Declarations

Declarations are statements that communicate to the language translator which data types are involved in any manipulation or storage of data. This is typically necessary for variables and functions.
Not all languages use declarations, in such languages variables are sometimes said to be typeless.

2.5.2.1 Purposes for declarations

1. Declarations allow *static* rather than *dynamic* type checking, meaning that the compiler can check for type errors at compile time. The extent to which a programming language can prevent type errors is referred to as its type safety. There are varying definitions of type safety. In one definition a function $f : S \to R$ is called type safe if f cannot generate a value outside of R.
2. They also allow the translator to determine what the best way to store data is.
3. Polymorphic disambiguation.

2.5.3 Type systems

A type system is a set of rules that govern how the language deals with types. The type system of each language is different, but classifications can be made based on certain aspects of the type systems.

2.5.3.1 Strong and weak typing.

Saying a language is strongly typed is a colloquial way to say that it has relatively stricter typing rules. Some authors require that all type errors can be detected statically (at compile time). In general the presence of type safety, memory safety, static type-checking or even dynamic type-checking can contribute to a language being called strongly typed.
The opposite of strong typing is weak typing.

2.5.3.2 Static vs dynamic.

Static type-checking is the process for verifying type safety of a program based on analysis of the source code. This can happen at compile time. Static type-checking can be considered a limited form of program verification. It can also (to some extent) reduce the need for dynamic type-checking, which makes the compiled program smaller and faster.
A number of useful and common programming features cannot be checked statically, so many languages have both static and dynamic type-checking. Many languages also provide ways to bypass the type checker.
Dynamic type-checking is the process for verifying type safety of a program at runtime. Dynamic typing has several advantages, including making metaprogramming (treating other programs as data) easier and allowing duck typing, making code reuse easier.

2.5.3.3 Manifest vs inferred.

Manifest typing is explicit identification by the programmer of the type of each variable being declared.
In contrast implicit typing is where the type of a data type is automatically detected.

2.5.3.4 Nominal vs structural vs duck typing.

In a structural type system (or a *property-based system*) elements are considered to be compatible with one another if the (relevant part of the) structure of the types are compatible.
This contrasts with duck typing where compatibility between types is only tested at runtime. The name comes from the duck test: "If it walks like a duck and quacks like a duck, it must be a duck".
Nominal typing determines compatibility and equivalence of data types based only on the explicit declarations and / or names of the types.

2.5.3.5 Type conversion and coercion

If there is a mismatch between the type of an argument and the expected type for that operation, the mismatch may be flagged as an error, or a coercion (i.e. an implicit type conversion may be applied.

2.5.3.6 Other concepts

Dependent types depend on a value. In intuitionistic type theory, dependent types are used to encode logic quantifiers.

Flow-sensitive typing can update a type to a more specific one if it follows a statement that validates its type.

Gradual typing allows software developers to choose whether to provide types explicitly, enabling static type-checking, or leave variables untyped, relying on dynamic type-checking.

Latent typing associates types with values and not variables. This typically requires runtime type checking and so is commonly used synonymously with dynamic typing.

Refinement types are types with a predicate that holds for any element of the refined type.

Unique types may have at most a single reference.

2.5.4 Type definitions

There are multiple ways to define new types:

1. We can define the new type as a compound type containing data of different types. For example we might define complex numbers as a new type containing two floats: the real and imaginary parts.

2. Subtypes have values that form a subset of a larger class of values. For example, in C *int* is a subtype of *long*.

2.6 Name resolution

There are two ways a data object can be made available as an operand for an operation:

1. *Direct transmission,* in which data is passed directly from the output of one operation into the input of the next. The may never receive a name.

2. *Referencing through a named object.* A data object may be given a name, which is then subsequently used to reference the data. Determining which data a specific name refers to is called name resolution.

   If the name resolution is performed at compile time, it is called **static name resolution**, if it is performed at runtime, it is called **dynamic name resolution**.

   Depending on the language, there may be different types of names:

   (a) Variable names.
   (b) Formal parameter names.
   (c) Subprogram names.
   (d) Names for defined types.
   (e) Names for defined constants.
   (f) Statement labels (names for statements).
   (g) Exception names.
   (h) Names for primitive operations (e.g., $+, -, *, \ldots$).
   (i) Names for literal constants (e.g., $42, 3.14$, "Hello world", ...).

2.6.1 Referencing environments

The referencing environment or refers to the set of identifier associations available for use during the execution of a program or subprogram. It may have several components:

1. The *local referencing environment* is the set of associations created in a subprogram and only available inside that subprogram.

2. The *nonlocal referencing environment* is the part of the referencing environment that is available inside a subprogram, but was not created inside it.

3. The *global referencing environment* is the part of the referencing environment available in a subprogram that was created in the main program. It is part of the nonlocal referencing environment.

4. The *predefined referencing environment* is defined in the language definition.

From the above distinctions it may be inferred that not all associations for identifiers are always visible within all parts of a program.
A data object may have different names, i.e. there may be different references to it, possibly in different referencing environments. If a data object has different names in the same referencing environment, these names are called aliases. This can make understanding program execution more difficult.

2.6.2 Namespaces

Modern programs use a lot of identifiers. It can be difficult to make sure there a no conflicting uses of any particular name. Namespaces make it possible for an identifier to have different meanings depending on its associated namespace.

2.6.3 Dynamic and lexical (static) scope

The scope of an association (i.e. name binding) is the part of the computer program during which it is part of a referencing environment.
If "part of the program" refers to the portion of the source code, it is known as the lexical scope. If on the other hand it refers to the "portion of runtime (i.e. time period during execution)", it is known as the dynamic scope.
There can be different levels of scope:

1. Expression
2. Block
3. Function
4. File
5. Module
6. Global

2.6.4 Overloading

Overloading makes it possible for an identifier to have different meanings depending on how it is used, even in a single namespace or scope.
The typical example is an identifier that points to different functions depending on the type of input it receives.

2.7 Memory management

2.7.1 Garbage collection

TODO

2.7.2 Stack and heap

TODO

2.8 Sequence control

2.8.1 Sequencing in expressions

2.8.1.1 In arithmetic expressions

Precedence rules traditionally dictate the resolution of many semantic ambiguities when working with infix notation.

Prefix (or Polish) notation is unambiguous and does not require parentheses. The stack-based notation was introduced by Polish mathematician Lukasiewicz, whose name nobody could pronounce, hence the name Polish. Postfix (or reverse Polish) notation has similar advantages.

Infix notation is clumsy for operations that are not binary.

Short-circuiting in Boolean expressions is when part of the expression is not evaluated because it does not influence the value of the whole expression. For example if the first argument of the and operator evaluates to false, the entire expression must be false. To be more efficient, the program might then not evaluate the second argument. This could be a problem if evaluating the second argument would have produced some side-effects (explained later) that were necessary.
Similarly if the first argument of the or operator evaluates to true, the second argument may not be evaluated.

2.8.1.2 Backtracking

2.8.2 Sequencing between statements

There are three main forms of statements-level sequence control. The structured program theorem (due to Böhm and Jacobini) states that they are sufficient to express any computable function. (The proof simply consists of the construction of the operations of a Turing machine using control structures). This is the theoretical basis of the structured programming paradigm, which aims to eliminate the goto statement.

1. **Compositions**. Statements are placed in the order they are supposed to be executed in.
2. **Alternation**. Statements may be alternatives so that one or the other is executed, but not both.
3. **Iteration**. A sequence of statements may be executed repeatedly.

2.8.2.1 Flow control statements.

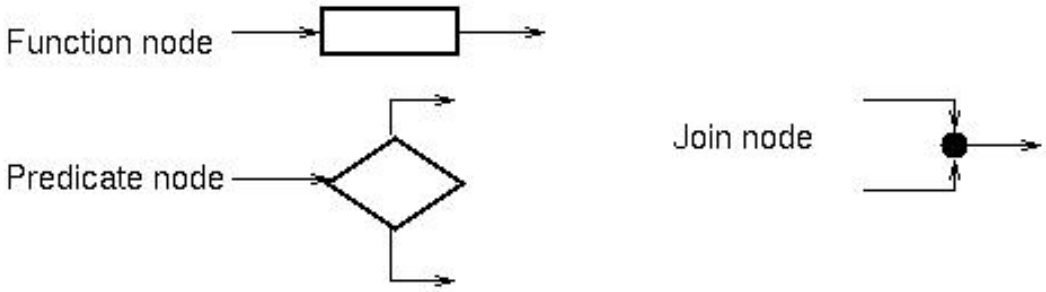
Some ways to control the sequencing of statements include:

- **Goto statement**. A goto statement instructs the computer to jump to a particular statement. Goto statements can be conditional or unconditional.

- **Conditional statements**
  - If statements
  - Case statements
- **Iteration statements.**
  - Simple repetition. A block of statements is repeated a number of times.
  - Repetition while a condition holds.
  - Repetition while incrementing a counter.
  - Indefinite repetition.
- **Break statement.** The break statement breaks out of the current loop.
- **Continue statement.** The continue statement moves execution on to the next iteration of the loop.
- **Return statement.** The return statement terminates the execution of a subroutine.

2.8.2.2 Flow chart representation and prime programs.

We can represent the flow of programs using flow charts with three types of nodes, see figure **??**.

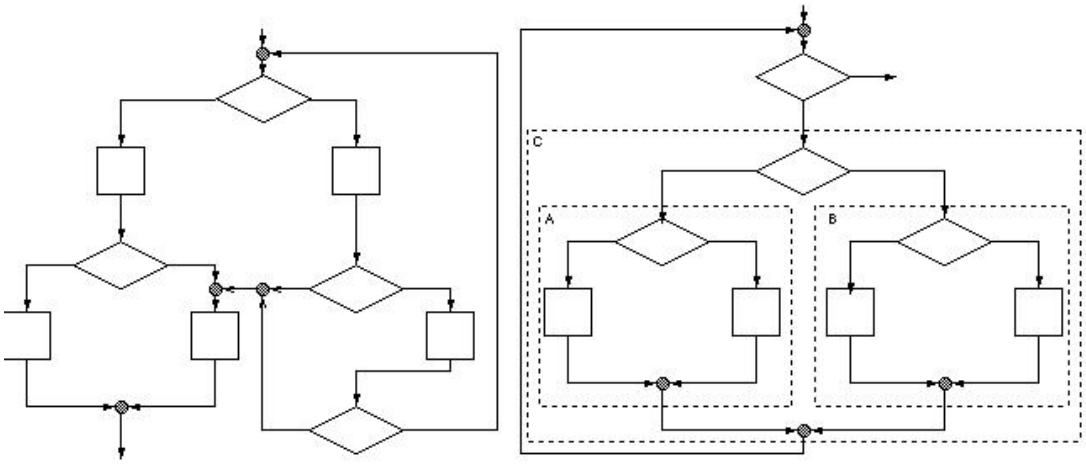


We define a proper program as a flowchart which:

1. has a single entry arc;
2. has a single exit arc; and
3. has a path from the entry arc to each node and from each node to the exit arc.

A prime program is a proper program which has no embedded proper subprogram greater than one node. A sequence of function nodes is also considered prime.
A composite program is a proper program that is not prime.
We can enumerate the prime programs. Figure **??** describes all programs of up to 4 nodes.
We can now classify these primes.

- Primes (a), (b), (e) and (i) represent sequences of function nodes.



(a) Prime Program (b) Composite Program

- Primes (c), (d), and (l) through (q) do not contain function nodes and thus do not change the state of the machine. They are either identity functions of or infinite loops.
- The primes (f), (g), (h), (j) and (k) are the interesting primes. They represent the following control structures:

(f) is the **if-then**,

(g) is the **while-loop**,

(h) is the **do-while**,

(j) is the **if-then-else** and

(k) is the **do-while-do**.

Most programming languages implement these control structures, except the do-while-do. This last structure has generally been ignored by language designers.
Other types of loops, such as for-loops can be considered syntactic sugar.

2.8.2.3 Structured programming.

The goto statement is quite controversial. The paradigm of structured programming aims to improve clarity by eliminating goto statements and instead making use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.
The goto statement has disadvantages:

1. The lack of hierarchical program structure can make programs less clear.
2. Making the order of statements in the program not necessarily the execution order can lead to confusion.
3. It violates the concept of *one-in, one-out control structures*, making the design less understandable.

4. Groups of statements may serve multiple purposes. (On the other hand it can get rid of code duplication.)

5. It makes it hard to find a meaningful set of coordinates with which to describe the process progress. (See Dijkstra.)

There are, however, also some advantages:

1. Direct hardware support. (This is relevant with poorly implemented Boolean variables and procedure calls, see in particular tail call optimization). TODO ref Knuth.

2. Simple and easy to use in small programs

3. Completely general purpose as a building block for simulating other control structures.

4. Some program structures are expressed more elegantly with a goto statement than with structures sequence control. Examples include:

   - Loops with multiple natural exit points.
   - Simulating do-while-do loops.
   - Prematurely exiting a loop to deal with exceptional conditions.
   - When implementing state machines, state switches may conveniently be implemented using goto statements. This type of state-switching is often used in the Linux kernel.

   In some languages these issues are addressed by using break statements, continue statements and throwing exceptions. All three of these however can be viewed as just a repackaging of the goto statement, with many of the same flaws.

   Having said that, the practical consensus seems to be that these constructs are useful in practice.

2.8.2.4 Continuations

First-class control TODO?

2.9 Subprogram control

2.9.1 Subprogram sequence control

2.9.1.1 Simple call-return.

In this view the main program may during execution call various subprograms, which may in turn call sub-subprograms. A very naive way to implement this is using the *copy rule*: the code of the subprogram is simply copied into the main program whenever it is called.
Limitations of this naive view include the following:

1. *Subprograms cannot be recursive.* A subprogram is directly recursive if it calls itself at some point. A subprogram is indirectly recursive if it calls another subprogram that in turn calls the original subprogram.

   If the copy rule were to be applied to a recursive subprogram, the resulting code would still call that subprogram. Thus we would be stuck in an infinite loop. This is problematic because recursion can be quite useful.

2. *Subprograms must execute completely at each call.* Sometimes, however, we want the subprogram to resume execution from where it left off last time. This is essentially the idea behind coroutines.

3. *Immediate transfer of control at point of call.* For a scheduled subprogram call we wish execution of the subprogram to be deferred until some later time.

4. *Single execution sequence.* This implementation of subprograms does not support parallel programming.

5. *Subprograms must be explicitly called.* Sometimes we do not know when we will want to call a subprogram. For instance an exception handler should be called whenever an exception is thrown. This is by definition exceptional.

Implementation First we make a distinction between the *definition* of a subprogram and its *activation*. An activation record contains a reference to the code of the subprogram as well as an *activation record* containing local data, parameters and other data.
Program flow is controlled by keeping track of pairs of pointers: The current-instruction pointer (CIP) points to the statement currently being executed and the current-environment pointer (CEP) points to the current activation record.
When a subprogram is called, the old CIP and CEP are stored somewhere. A new CIP is created pointing to the first statement of the code of the subprogram. A new CEP is created pointing to the newly created activation record.
When a subprogram ends, the old CIP and CEP are reinstated.
There are multiple ways the CIP and CEP may be stored. A stack may be used. Or they may be stored in the activation record of the new subprogram. If we only require the simple copy rule behaviour, we can even get away without the CEP: each subprogram can only ever have at most one activation record and thus we can statically allocate storage for a single activation record.
This simple model is often hardware-supported with a *return-jump instruction*.

2.9.1.2 Recursion.

So long as both the CIP and CEP are stored, the implementation idea given above automatically supports recursion.

2.9.1.3 Exceptions.

These are raised to signal exceptional circumstances, such as

1. Error conditions

2. Unpredictable conditions

3. Tracing and monitoring

An exception is typically propagated back up the call stack until the proper place th handle it. The question of what to do once an exception has been handled does not have an obvious answer. Different languages have different solutions.

Assertions. An assertion is a statement implying a relation among data in a program. During testing it will typically raise an error if that relation is violated. After testing it may remain as documentation. See also design by contract.

2.9.1.4 Scheduling.

Scheduling can take several different forms, such as

1. Schedule subprograms to execute before or after other subprograms.
2. Schedule subprograms the be executed when a Boolean expression becomes true.
3. Schedule the execution of subprograms based on time.
4. Schedule subprograms based on priority.

2.9.1.5 Coroutines

. Coroutines are subprograms that can exit before completing execution. When it is called again, it resumes from where it left off. Two coroutines can pass control between themselves.

2.9.1.6 Generators

. Generators, also known as semicoroutines, are like coroutines, but lack a coroutines ability to specify where it yields control to. It is still possible to implement coroutines on top of a generator facility, with the aid of a top-level dispatcher routine.

2.9.1.7 Parallel programming

TODO: move to separate section?

Things to take into account. Parallel programming constructs add complexity to the language design. The following topics must be adressed.

1. *Variable definitions.* If variables are mutable, we can run into synchronization problems. Definitional (i.e. immutable) variables do not have this problem.
2. *Parallel composition.* We need ways to fork program execution and create additional threads of control.
3. *Program structure.* Parallel programs generally follow one of two execution models:
   (a) They may be transformational, where the goal is to transform the input data into an appropriate output value. Parallelism is applied to speed up the process.
   (b) They may be reactive where the program reacts to external stimuli, called events.
4. *Communication* between parallel programs may happen through the use of shared memory or via messages.

And or fork statements With this statement we can designate parts of the program that we can allow to be executed at the same time.

Guarded commands These were proposed in the 1970s by Dijkstra. They provide a way to write nondeterministic programs.
A guarded statement has a condition, called a guard, such that the statement is not executed if the guard is false. We consider the following types of guarded statements:

- The **guarded if statement** contains several guarded statements. At least one for the guards must be true. If multiple guards are true, the statement to execute is chosen nondeterministically.
- The **guarded repetition statement** is like the guarded if statement that repeats as long as some guard is true.

TODO: move?? Also expand.

Tasks Ordinarily when a subprogram is called, execution of the main program is suspended. If the subprogram is called as a task, the execution of the main program continues.
Task management can obviously be non trivial. In order for tasks running asynchronously to coordinate their activities, the language must provide some means of synchronization. This may be achieved in several ways:

1. **Interrupts**. If task A wishes to signal to task B that a particular event has occurred, then task A executes an instruction that causes the execution of task B to be interrupted immediately. Control is then handed to a subprogram or section of code that handles the interrupt. Afterwards control is handed back to task B.

   This is a mechanism commonly found in computer hardware. It is also similar to exception handling. In high level-languages, interrupts have several disadvantages:

   (a) The code for interrupt handling is separate from the main body of the task, leading to a more confusing program structure.

   (b) A task waiting for an interrupt must usually enter a busy waiting loop.

   (c) The task must be written so that an interrupt can be handled at any time.

2. **Semaphores**. TODO. A semaphore is a data object consisting of two parts:

   (a) An integer counter. In a binary semaphore this counter may only be one or zero, in a general semaphore it may be any non-negative integer.

   (b) A queue of tasks.

   Two operations are defined for a semaphore data object P:

   (a) `signal(P)`. When executed by a task A, this operation tests the value of the counter in P. If zero, the first task in the task queue is removed from the queue and executed. If not zero, or if the queue is empty, the counter is incremented by one. Execution of task A continues after the signal operation is complete.

   (b) `wait(P)`. When executed by task B, this operation tests the value of the counter in P. If nonzero, the counter is decremented by one and task B continues execution. If zero, task B is inserted at the end of the task queue for P and the execution of B is suspended.

   Semaphores have some disadvantages for use in high-level languages:

   (a) A task can only wait for one semaphore at a time.

(b) If a task fails to signal, the system of tasks may deadlock.

(c) Programs with several tasks and semaphores become increasingly difficult to understand, debug and verify.

(d) All tasks accessing the semaphore must share memory.

3. **Messages** can be passed between tasks. The basic concept is similar to a pipe. Implementations of this system can be quite complex. Several tasks may simultaneously try to send messages. The implementation must then provide some way to store those messages until the receiver can process them.

4. **Guarded commands.** See above.

5. **Rendezvous.** Used in Ada. Like messages, but must be accepted.

2.9.2 Shared data in subprograms

Data can either be shared directly, using parameters, or indirectly by giving the subprogram access to nonlocal environments.

2.9.2.1 Parameters and parameter transmission

A (formal) parameter is a variable declared in the function definition, that must be given some data when the function is called. This data that is given to the function is called an argument or actual parameter.
There are multiple ways to specify which argument should go to which parameter. This can, e.g., be done through positional correspondence or by explicitly naming the parameter.
There are also multiple ways to transmit parameters. Depending on the method of transmission, parameters can also be used to return data from the subprogram. In most languages a single value may also be returned as an explicit function value.
In some languages the distinction is made between between function calls with a return value and procedure or subroutine calls without a return value.

Call by name. In this model the actual parameter is substituted everywhere for the formal parameter. This results in overhead leads to complications.

Call by reference. A pointer to the location of the data object is made available to the subprogram.

Call by value. The value of the argument is copied into a variable with the name of the formal parameter.

Call by value-result. Like call by value, except at the end of the subprogram the contents of the local variable is copied back into the original argument variable.

Call by constant value. The formal argument acts like a local constant.

Call by result. This parameter is only used to transmit a result back from a subprogram.

2.9.2.2 Data sharing through environments.

TODO

Explicit common environments. TODO

2.9.3 First-class subprograms

If we say a language has first-class subprograms or functions, it means functions are treated just like any other data type. TODO

2.9.3.1 Closures

TODO

2.9.4 Factors that obscure the definition of subprograms as mathematical functions

1. Implicit arguments.
2. Implicit results (side-effects).
3. Behaviour can be undefined for some arguments. In general the domain is too broad and needs to be narrowed with conditional statements.
4. History sensitive.

2.9.5 Tail call optimization

TODO

2.10 Miscellaneous language features

TODO Reflection

2.11 Tools

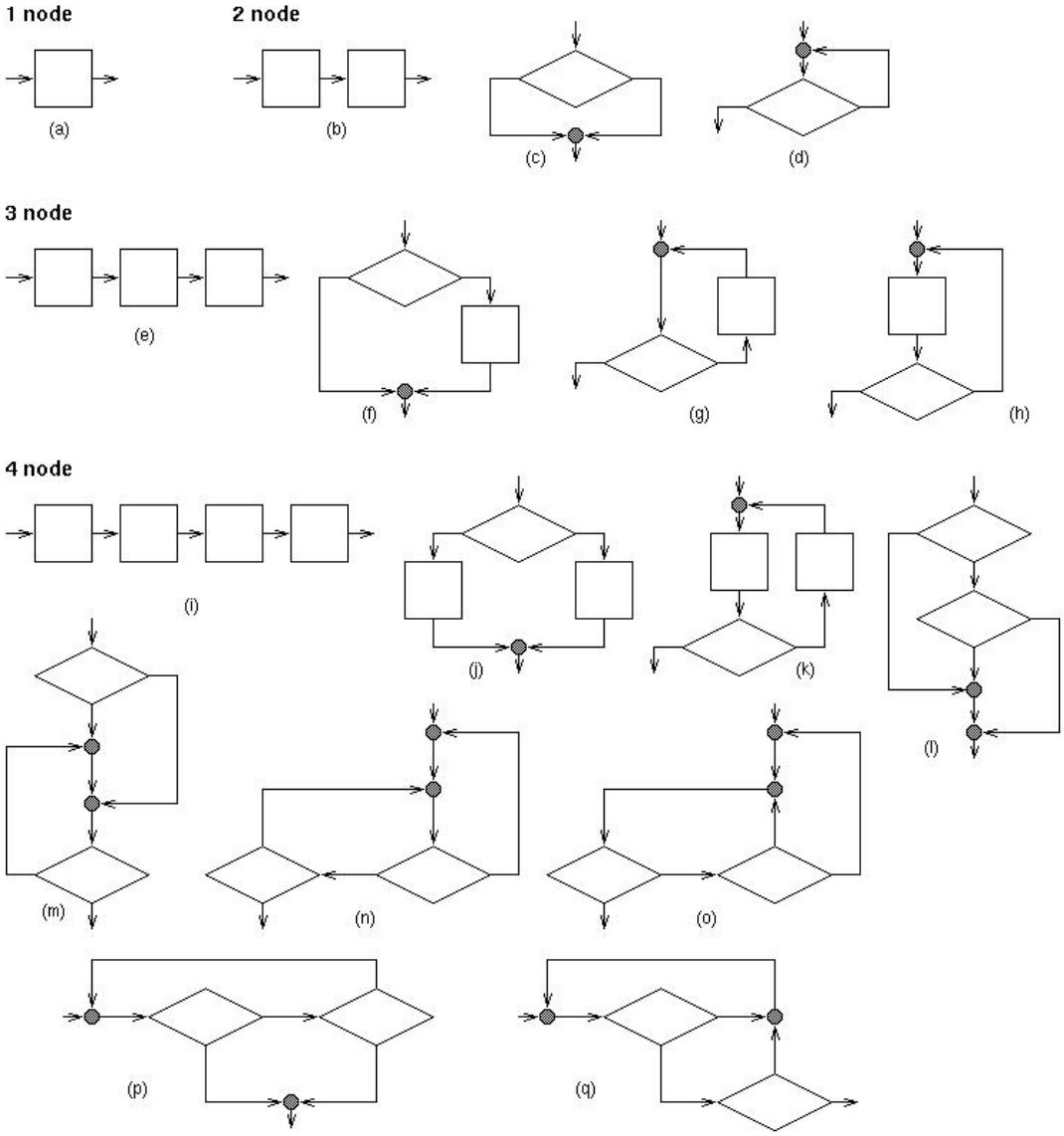
2.11.1 ANTLR

2.11.2 YACC - LEX / BISON

2.11.3 LLVM

Also JVM, GCC?

2.11.4 Racket


1 node
2 node
(a)
(b)
(c)
(d)
3 node
(e)
(f)
(g)
(h)
4 node
(i)
(j)
(K)
(l)
(m)
(n)
(o)
(p)
(q)

Chapter 3

Paradigms and patterns

3.1 The object-oriented paradigm

3.1.1 Encapsulation

3.1.2 Inheritance

3.1.3 Polymorphism

3.2 Design patterns

Observer, Decorator, Factory, Singleton, Command, Adapter, Facade, Template method, Iterator, Composite, State, Proxy, Compound patterns
Bridge, Builder, Chain of responsability, Flyweight, Interpreter, Mediator, Memento, Prototype, Visitor
Visitor, listener, module

3.2.1 Dynamic languages

3.2.1.1 Monkey patching

Chapter 4

Practical coding

code smell
readability (clarity)
Magic numbers
SOLID, STUPID
DRY
Object calisthenics:

1. Only one level of indentation per method
2. Do not use else
3. Wrap primitive types (especially if they contain behavior)
4. Only one method call per line.
5. Do not abbreviate
6. Keep classes small: about 10 methods of 10 lines each.
7. Limit your instance variables to two.
8. Use first class collections
9. Don't use getters and setters. Too generic.

4.1 Development processess

4.1.1 Iterative and waterfall processes

The essential difference is how a project gets broken up into smaller chunks. To build software you might have to do certain activities: requirements analysis, design, coding and testing.

- The waterfall style breaks down the project based on those activities, e.g. having 2-month analysis, 4-month design, 3-month coding and 3-month testing phases.

- The iterative styles breaks down a project by subsets of functionality. This means performing all four activities so you have a systems that works well with only a part of the functionality, then performing all four activities again to add more functionality. And again, and again.

  A common technique is time boxing where the scope of each iteration is determined by time, not feature set.

  At the end of each iteration an **iteration retrospective** may be held. A good way to do this is to make a list with three categories:

  - *Keep:* things that worked well that you want to ensure you continue to do.
  - *Problems:* areas that aren't working well.
  - *Try:* changes to your process to improve it.

In practice, of course, there is some overlap. With waterfall development there are often backflows: during coding and testing it may be necessary to revisit analysis and design. These backflows should be minimised as much as possible.
With iteration there is usually some form of exploration activity and analysis before the true iteration begins. At the end there is usually a global stabilisation period to iron out bugs. Also other activities, such as user training, may not be part of the iteration.
There are also hybrid processes, such as first doing analysis and high-level design in waterfall style and then dividing coding and testing into iterations.

> Many people agree pure waterfall is bad, but it is still most common in industry. Often people claim to be working iteratively when in fact they doing waterfall. Common symptoms include
>
> - "We are doing one analysis iteration followed by two design iterations."
> - "This iteration's code is very buggy, but we'll clean it up at the end."

TODO: rework. The iterative approach explicitly assumes that existing code will be reworked and deleted. This process can be made more efficient with

- Automated regression testing
- Refactoring
- Continuous integration

4.1.2 Predictive and adaptive planning

With predictive planning a project has two phases. The first is coming up with plans and is difficult to predict, but the second is much more predictable because the plans are in place.
The problem is that often requirements change. This is called requirements churn. To combat this you could spend more time planning, but often requirements churn is unavoidable. In such cases one needs to settle for adaptive planning, where everything is constantly reevaluated.

4.1.3 Agile processes

The Manifesto for Agile Software Development (`https://agilemanifesto.org/`) states the following values:

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan

As well as the following principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile processes are strongly adaptive and very much people-oriented. They also tend to use short, time-boxed iterations with little weight attached to documents and ceremony. For this reason they are often characterised as **lightweight** (this is a consequence of adaptivity and people orientation).

4.1.3.1 Extreme programming (XP)

4.1.3.2 Scrum

4.1.3.3 Feature driven development (FDD)

4.1.3.4 Crystal

4.1.3.5 Dynamic systems development method (DSDM)

4.1.4 Rational unified processes (RUP)

RUP is a project framework: the first step is to choose a development case, i.e. the process used in the project.

All RUP projects follow four phases:

- Inception
- Elaboration
- Construction
- Transition

RUP is only compatible with an iterative work flow.

4.2 Design documents and documentation

Detailed documentation generated from the code and external documentation. External documentation should be comprehensible, not comprehensive.
Document design alternatives that were not chosen and why not.

4.3 Coding styles

defensive, total, design by contract

4.4 Testing

Difference in style

4.4.1 Unit testing

4.5 Coding in a team

I have very little experience with this. This section is mostly just a place for me to put tips I have heard for future reference.

4.6 Version control

4.6.1 Git

4.6.2 Subversion

4.7 Benchmarking

4.8 Optimization

Precompute (integral pictures)
Only worry about bottlenecks. Premature optimization is the root of all evil.
Inner loops
Timing analysis

Chapter 5

Notes on selected languages

5.1 History

Here an attempt will be made to very briefly highlight some of the ideas, features and syntax of a number of languages. The classification is quite arbitrary as most languages support multiple paradigms.

5.2 Assembly languages

RISC ¿¡ CISC

5.2.1 M32

5.2.2 Z80

5.2.3 (M)MIX

5.2.4 x86

5.2.5 ARM

5.3 Procedural languages

5.3.1 FORTRAN 77

History. FORTRAN (FORmula TRANslation) was the first high-level programming language to become widely used. At conception, in 1957, there were serious concerns as to whether a high-level language could ever seriously compete with assembly languages. Thus design was oriented heavily towards efficiency.
The language went through many revisions. The biggest change was between FORTRAN 77 and Fortran 90. Fortran 95 is the most widely implemented of the more recent specifications and later versions are largely similar.

Syntactic features. The basic structure of a program can be seen in this hello world example.

```
1       program helloworld
2         print *, "Hello world!"
3         stop
4         end
```

The `stop` statement is optional since the program will stop when it reaches the end, but is recommended to emphasize that execution flow stops there. You cannot have a variable with the same name as the program.
FORTRAN is completely case insensitive and was traditionally written in all caps. Current practice is to mix upper and lower case.
Blanks are ignored completely, in fact they may all be removed. Spaces may be inserted freely.
FORTRAN 77 uses fixed-format source code:

- Comments must begin with a `C` or a `*` in column 1. Some compilers allow in-line comment with an `!`. The exclamation mark may appear anywhere on a line, except in positions 2-6.
- Statement labels must occur in columns 1-5
- Continuation lines must have a non-blank character in column 6. Any non-blank character can be used as a continuation character. Traditionally a plus, an ampersand or a digit is used. Example:

```
c The next statement goes over two physical lines
      area = 3.14159265358979
     +       * r * r
```

- Statements must start in column 7.
- The line-length may be limited to 72 characters (derived from the 80-byte width of a punch-card, with last 8 characters reserved for (optional) sequence numbers)

Variables, types and declarations Variables are 1 to 6 characters long, begin with a letter and may contain letters and digits.
Reserved words are: `assign, backspace, block data, call, close, common, continue, data, dimension, do, else, else if, end, endfile, endif, entry, equivalence, external, format, function, goto, if, implicit, inquire, intrinsic, open, parameter, pause, print, program, read, return, rewind, rewrite, save, stop, subroutine, then, write`.
Variables do not have to be explicitly declared. Explicit declarations take the form

```
      integer A, B, C
      real radius, eulerE
      double precision q, r
      complex z
      logical t
      character ch
```

If there is no explicit declaration, a *naming convention* is used. By default variable names beginning with `I-N` are integer variables; all others are real. This default behaviour can be modified with an `IMPLICIT` statement:

```
      IMPLICIT INTEGER (A-Z)
```

This causes the type integer to be assumed for all variables.
Constants are defined in the following way

```
      PARAMETER (KMAX = 100, MIDPT = 50)
```

the `parameter` statement(s) must come before the first executable statement.

Arrays. One-dimensional arrays are declared as follows.

```
      real a(20)
      real b(0:19), weird(-162:237)
      double precision x(100)
```

By default arrays are indexed from 1 to their length, inclusive. Arbitrary index ranges can also be defined, see arrays `b` and `wierd` above.
Elements or arrays can be accessed using brackets: `a(3) = 2.0`. The Fortran compiler does not check whether array elements are out of bounds or undefined!
Multi-dimensional arrays can be declared in a similar way by putting a comma between indexes:

```
      real A(3,5,2:7)
      A(1,4,2) = 5
```

Two-dimensional arrays are stored in memory as contiguous sequences of elements, column by column.
An alternative (old-fashioned) style is to use the `dimension` statement:

```
      real A, x
      dimension x(50)
      dimension A(10,20)
C This is equivalent to:
      real A(10,20), x(50)
```

The `DATA` statement is a compact way to input data that is known at compile time. The syntax is as follows

```
      data m/10/, n/20/, x/2.5/, y/2.5/
C Or alternatively
      data m,n/10,20/, x,y/2*2.5/
```

Both statements result in `m = 10, n = 20, x = 2.5, y = 2.5`.
The data statement is performed only once, right before the execution of the program starts. For this reason, the data statement is mainly used in the main program and not in subroutines. It can also be used to initialize arrays (vectors, matrices).

```
      real A(10,20)
      data A/ 200 * 0.0/
C Individual elements may also be initialised
      data A(1,1)/ 12.5/, A(2,1)/ -33.3/, A(2,2)/ 1.0/
```

The data statement cannot be used for variables contained in a common block. There is a special syntax for this purpose, called `block data`.

```
      block data
        integer nmax
        parameter (nmax=20)
```

```
      real v(nmax), alpha, beta
      common /vector/v,alpha,beta
      data v/20*100.0/, alpha/3.14/, beta/2.71/
    end
```

The block data may not be nested inside the main program or a subroutine.

Expressions and assignment FORTRAN supports the following literals:

- Integer literals

```
      1
      0
      -100
      32767
      +15
```

- Real literals

```
      1.0
      -0.25
      2.6E6
      3.333E-1
```

  The E means multiply by 10 to the power of whatever comes after it.

- Double precision literals

```
      2.0D-1
      1D99
```

  Like reals, but D instead of E.

- Complex literals

```
      (2, -3)
      (1., 9.9E-1)
```

- Logical literals

```
      .TRUE.
      .FALSE.
```

- Character literals. Most often used in arrays (as strings). Some versions of FORTRAN require single quotes. An enclosed single-quote can be represented by doubling.

```
      'ABC'
      'Anything␣goes!'
      'It''s␣a␣nice␣day'
```

Operator precedence in FORTRAN 77 (from highest to lowest):

** Exponentiation

– Unary minus

`*, /` Multiplication, division

`+,-` Addition, subtraction

Relational operators: `.LT., .LE., .GT., .GE., .EQ., .NE.` (respectively $<, <=, >, >=, =, \backslash =$).

`.NOT.` Logical not

`.AND.` Logical and

`.OR.` Logical or

For different evaluation order, use parentheses.
Caution must be taken when using the division operator. If both operands are integer, integer division is performed, otherwise real arithmetic division is performed.
Assignment is done with the equals sign

```
      area = pi * r**2
```

For type conversion, the following functions are available: `int(), real(), dble(), ichar(), char()`. The function `ichar` converts a character into an integer; `char` does the opposite.
Constructs that are not statically checked are ordinarily left unchecked.

Flow control. The `if` statement comes in several forms:

```
      if (x .LT. 0) x = -x
```

or

```
      if (x .LT. 0) then
        x = -x
      endif
```

or

```
      if (x .LT. 0) then
        x = -x
      elseif (x .GT. 5) then
        x = 5
      else
        x = x+1
      endif
```

`if` statements can also be nested.
FORTRAN 77 only has the `do`-loop. An example:

```
      integer i, n, sum

      sum = 0
      do 10 i = 1, n
         sum = sum + i
         write(*,*) 'i␣=', i
         write(*,*) 'sum␣=', sum
 10   continue
```

Here the number 10 is a label. Column positions 1-5 are reserved for statement labels. Typically, most programmers use consecutive multiples of 10.
The `continue` statement is a placeholder that does nothing.
The statement identified by the label after the `DO` command is called the terminal statement.
The variable is typically incremented by one each loop until the second value is reached (inclusive). A step may also be specified (negative means counting down).

```
      integer i

      do 20 i = 10, 1, -2
         write(*,*) 'i␣=', i
 20   continue
```

Many compilers also allow the `DO` statement to be closed by `ENDDO`. This is not ANSI FORTRAN 77, however. The expressions in the beginning of the `DO`-loop arwe evaluated only once. So the following loop does not run forever:

```
      integer i,j

      read (*,*) j
      do 20 i = 1, j
         j = j + 1
 20   continue
      write (*,*) j
```

Other types of loops have to be simulated using `GOTO` statements.

```
     integer n

     n = 1
 10  if (n .LE. 100) then
        write (*,*) n
        n = 2*n
        goto 10
     endif
```

Subprograms. Fortran has two different types of subprograms, called functions and subroutines.
Functions take a set of input arguments (parameters) and return a value of some type. Some built-in functions include:

```
abs     absolute value
min     minimum value
max     maximum value
sqrt    square root
sin     sine
cos     cosine
tan     tangent
atan    arctangent
exp     exponential (natural)
log     logarithm (natural)
```

An example of a user-defined function:

```
      real function r(m,t)
        integer m
        real t
        r = 0.1*t * (m**2 + 14*m + 46)
        if (r .LT. 0) r = 0.0
        return
      end
```

We see

- Functions have a type. If none is declared, the same implicit typing system as for variables is used.
- Functions are terminated by the `return` statement.
- The return value should be stored in a variable with the same name as the function.
- Strictly speaking Fortran 77 does not permit recursion. However, it is not uncommon for a compiler to allow recursion.

A subroutine does not return a value. The syntax is as follows:

```
      subroutine iswap (a, b)
        integer a, b
        integer tmp
        tmp = a
        a = b
        b = tmp
        return
      end
```

The subroutine works because parameters in FORTRAN 77 subprograms are passed by reference. We can pass arrays of arbitrary size to subprograms if we define the arrays in the subprogram with dummy indices (usually an asterisk).

Common blocks and scope. FORTRAN 77 has no global variables. Common blocks can be used instead. In general the syntax is as follows

```
      program main
        real alpha, beta
        common /coeff/ alpha, beta
        [ statements ]
        stop
      end
C
      subroutine sub1 ( [ some arguments ] )
        [ declarations of arguments ]
        real alpha, beta
        common /coeff/ alpha, beta
        [ statements ]
        return
      end
```

```
C
      subroutine sub2 ( [ some arguments ] )
        [ declarations of arguments ]
        real alpha, beta
        common /coeff/ alpha, beta
        [ statements ]
        return
      end
```

The syntax of a common statement is: `COMMON / name / list-of-variables`. A variable cannot belong to more than one common block. The variables in a common block do not need to have the same names each place they occur (although it is a good idea to do so), but they must be listed in the same order and have the same type and size.
Putting arrays in common blocks is a bad idea.

I/O. In FORTRAN each file is associated with a unit number, an integer between 1 and 99. Some unit numbers are reserved: 5 is standard input, 6 is standard output.
Before you can use a file you have to open it. The command is

```
      open (list-of-specifiers)
```

Where the most common specifiers are

```
[UNIT=]  u
IOSTAT=  ios
ERR=     err
FILE=    fname
STATUS=  sta
ACCESS=  acc
FORM=    frm
RECL=    rl
```

`u` is a number between 1 and 99 inclusive that denotes the file.

`ios` is the I/O status identifier and should be an integer variable. Upon return, `ios` is set to zero if the statement was successful and set to a non-zero value otherwise.

`err` is a label which the program will jump to if there is an error.

`fname` is a character string denoting the file name.

`sta` is a character string that has to be either `NEW`, `OLD` or `SCRATCH`. It shows the prior status of the file. A scratch file is a file that is created when opened and deleted when closed (or the program ends).

`acc` must be either `SEQUENTIAL` or `DIRECT`. The default is `SEQUENTIAL`.

`frm` must be either `FORMATTED` or `UNFORMATTED`. The default is `UNFORMATTED`.

`rl` specifies the length of each record in a direct-access file.

After a file has been opened, you can access it by read and write statements. When you are done with the file, it should be closed by the statement

```
      close ([UNIT=]u[,IOSTAT=ios,ERR=err,STATUS=sta])
```

In this case `sta` is a character string which can be `KEEP` (the default) or `DELETE`.
Reading and writing is done as follows:

```
read ([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s) [ list-of-variables ]
write([UNIT=]u, [FMT=]fmt, IOSTAT=ios, ERR=err, END=s) [ list-of-variables ]
```

The `END=s` specifier defines which statement label the program jumps to if it reaches end-of-file.
The format number `fmt` refers to a label for a format statement (described below).
The first two arguments can be replaced with asterisks. This is sometimes called list directed read / write. This format assumes the file has a line per record and the fields are separated by blanks or commas.
If we are reading or writing to the stantard input, the followinf syntax may be used (the asterisk refers to the format):

```
      read  *, [ list-of-variables ]
      print *, [ list-of-variables ]
```

The `FORMAT` statement TODO

Language design notes Changes compared to FORTRAN 66:

- Additions:
  - Block `IF` and `END IF` statements, with optional `ELSE` and `ELSE IF`.
  - `DO`-loop extensions including
    - parameter expressions
    - negative increments
    - zero trip counts
    - Better file I/O.
    - The `IMPLICIT` statement.
    - The `PARAMETER` statement.
    - The `CHARACTER` datatype.
    - Lexical comparison of strings with `LGE, LGT, LLE, LLT`.
- Deprecated:
  - Hollerith constants and Hollerith data.
  - Transfer of control out of and back into the range of a DO loop (also known as "Extended Range")

5.3.2 Fortran 95

Changes compared to Fortran 90:

- Additions:
  -
- Deprecated:

–

- Changes:

–

Changes in FORTRAN 2003:

- Additions:

–

- Deprecated:

–

- Changes:

–

Changes in FORTRAN 2008:

- Additions:

–

- Deprecated:

–

- Changes:

–

Changes in FORTRAN 2018:

- Additions:

–

- Deprecated:

–

- Changes:

–

5.3.3 C

5.3.4 Pascal

Of course, in stupid languages like Pascal, where labels cannot be descriptive, GOTOs can be bad. But that's not the fault of the goto, that's the braindamage of the language designer.

5.3.5 SNOBOL

5.3.5.1 History

Development of SNOBOL (StriNg Oriented and symBOlic Language) began in 1962 at AT&T Bell Labs by Ralph Griswold, Ivan Polonsky and David Farber. The goal was to develop a string processing language for formula manipulation and graph analysis. The most successful version was SNOBOL4, developed in the 1970s.
SPITBOL (Speedy Implementation of SNOBOL) is a compiled implementation of the SNOBOL4.

5.3.5.2 Concepts

- Pattern matching language based upon BNF Grammars.
- Patterns are a first-class data type.

5.3.6 Go

5.4 UML

5.4.1 What is it?

The Unified Modeling Language (UML) is a family of graphical notations that help describing and designing software systems, particularly software systems built using the object-oriented paradigm. The UML is a relatively open standard, controlled by the Object Management Group (OMG), an open consortium of companies. The UML was born out of the unification of many graphical modeling languages around in the late 1980s and early 1990s.

5.4.1.1 Ways of using it

One can identify three modes in which UML is often used:

1. **UML as a sketch.** In this usage UML is used to communicate some aspect of a system. Such a sketch is by no means complete, but just meant to illustrate an idea. Selectivity is key.
2. **UML as a blueprint.** In this usage the UML diagram is meant to be a complete and detailed guide to the code. A blueprint may be of the whole system or of a particular area. Such diagrams can be used by specialised CASE (computer aided software engineering) tools.
3. **UML as a programming language** is when the blueprint is complete and accurate enough that it can mechanically be converted into executable code. The standard approach is called Model-Driven Architecture (MDA).

The UML can also be used for conceptual modeling or software modeling.

1. In the **software perspective**, the UML elements map (fairly) directly to elements in the software system. Within the software perspective, a distinction can also be made between diagrams focusing on *interface* or *implementation*.
2. In the **conceptual perspective**, the diagram deals in concepts, not necessarily actual components in the system.

The UML standard can also be seen as having either *prescriptive* or *descriptive* rules. TODO Fowler p.13

5.4.1.2 Notation and meta-model

The UML defines

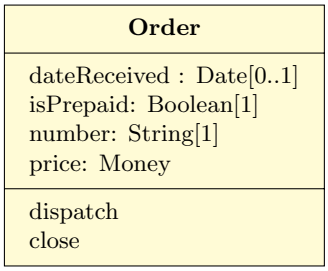
- a **notation** (i.e. graphical syntax) and
- a **meta-model** which defines the concepts of the language.

When referring to the UML, sometimes the notation and sometimes the meta-model is meant.

5.4.1.3 UML diagrams

5.4.2 Class diagrams

A class diagram describes the various types of objects in the system and the static relationships that exist among them. Each class has a name and several features, i.e. the properties and operations of a class.



5.4.2.1 Properties

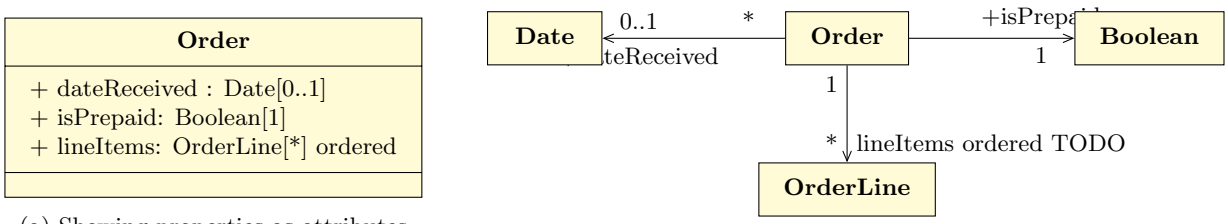
Properties represent the structural features of a class. Properties appear in two very distinct notations: *attributes* and *associations*.

Attributes define the property within the class box itself. They are put in the section under the title. The full form of an attribute is:

(*visibility***)?** **(/)?** *name* **(:** *type***)?** **([***multiplicity***])?** **(=** *default***)?**
(*property-string***)?**

Only *name* is necessary.

- *visibility* can be
  - \+ for public
  - \- for private
  - $\sim$ for package
  - \# for protected
- / means the property is a derived property, i.e. calculated from other properties.
- *name* is the name of the attribute.
- *type* indicates what kind of object may be placed in the attribute.
- *multiplicity* shows how many objects may fill the property. Multiplicities may be specified as
  - *number*: the property contains *number* objects.
  - *lowerBound*..*upperBound*: the property contains more than *lowerBound*, but no more than *upperBound*, objects.
  - *: the property contains an arbitrary number (zero or more) of objects.



(a) Showing properties as attributes

(b) Showing properties as associations

Figure 5.1: The same properties represented in two different notations

  - *lowerBound*`..*`: the property contains more than *lowerBound* objects.

- *number* is the value for a newly created object if the attribute is not specified during creation.

- *property-string* is of the form

  { *property-modifier* **(**, *property-modifier***)*** }

  where *property-modifier* can be one of

  - `id`: the property is part of the identifier for the class which owns the property.
  - `readOnly`: clients may not modify the property.
  - `ordered` or `unordered`: the objects associated with this multi-valued property are (un)ordered.
  - `unique` or `nonunique`: the objects associated with this multi-valued property may or may not have duplicate values.
  - `sequence` (or `seq`): `ordered` and `nonunique`.
  - `bag`: `unordered` and `nonunique`.
  - `union`: the property is a derived union of its subsets TODO?.
  - `redefines` *property-name*: the property redefines an inherited property named *property-name*.
  - `subsets` *property-name*: the property is a subset of the property named *property-name*.
  - *property-constraint*: A constraint that applies to the property.

Associations are another way to notate properties. An association is a solid line between two classes, directed from the source class to the target class. All extra information about the property is written at the target end of the association.
In general it is useful to use attributes for small things and value types (TODO ref), such as dates and booleans, and associations for more significant classes.

5.4.2.2 Operations

5.4.3 Sequence diagrams

5.4.4 Executable UML and model-driven architecture

5.5 Object-oriented languages

5.5.1 Ada

5.5.2 C++

5.5.3 Smalltalk

5.5.4 Java

5.5.5 Scala

History Released in 2004 by Martin Odersky. Provides support for functional programming. Designed to be compiled to Java bytecode, so that Scala applications can be executed on a Java Virtual Machine (JVM).

Basic setup Scala has a tool called the REPL (Read-Eval-Print Loop) that is analogous to commandline interpreters in many other languages. You may type any Scala expression, and the result will be evaluated and printed.
Results are saved as values and may be used in future expressions.
REPL sessions can be saved and loaded using `:save [path]` and `:load [path]`.
Recent history can be shown with `:h?`.
Full programs have the following form. The entry point is defined using an object with a single method, `main`.

```
1 object Application {
2   def main(args: Array[String]): Unit = {
3     // stuff goes here.
4   }
5 }
```

I/O. Printing can be done with `println("Hello world!")`, which forces a newline on next print, or with `print("Hello world!")`, which does not.
To read a file line by line

```
import scala.io.Source
for(line <- Source.fromFile("myfile.txt").getLines())
  println(line)
```

To write a file using Java's `PrintWriter`

```
val writer = new PrintWriter("myfile.txt")
writer.write("Writing␣line␣for␣line" + util.Properties.lineSeparator)
writer.write("Another␣line␣here" + util.Properties.lineSeparator)
writer.close()
```

```
// --- Importing things
import scala.collection.immutable.List
// --- Import all "sub packages"
import scala.collection.immutable._
// --- Import multiple classes in one statement
import scala.collection.immutable.{List, Map}
// --- Rename an import using '=>'
import scala.collection.immutable.{List => ImmutableList}
// --- Import all classes, except some. The following excludes Map and Set:
import scala.collection.immutable.{Map => _, Set => _, _}
// --- Java classes can also be imported. Scala syntax can be used
import java.swing.{JFrame, JWindow}
```

Syntactic elements Comments:

```
// Single line comment
/*
Multiline
comment
*/
```

Scala is strongly typed. Types can be checked without evaluating the expression. (This can be done in the REPL with `:type`).

Blocks Expressions can be combined by surrounding them with braces {}.

```
println(7) // Prints 7

println {
  val i = 5
  i + 2
} // Prints 7
```

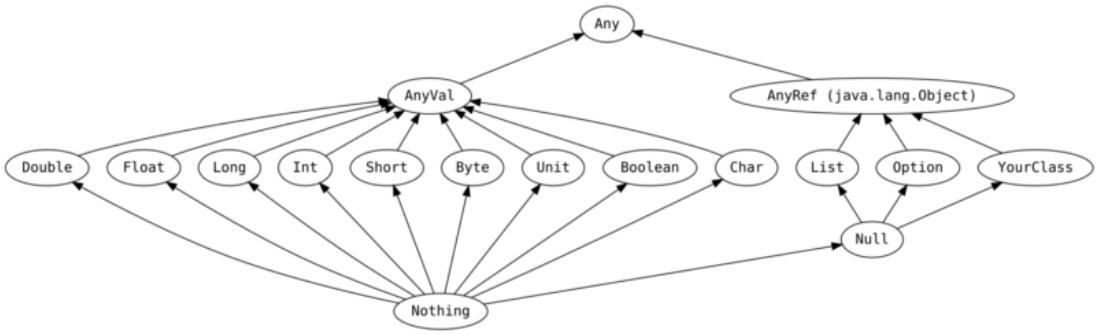
Values, variables and data types Values are like immutable variables, variables are mutable. Declaration is done with the `val` and `var` keywords, respectively.

```
val x = 10 // x is now 10
x = 20     // error: reassignment to val
var y = 10
y = 20     // y is now 20
```

Even though Scala is a statically typed language, explicit type declaration is often not needed, due to *type inference*. Explicit type declarations can be done as follows:

```
val z: Int = 10
val a: Double = 1.0
val b: Double = 10
```

Scala is considered a pure object-oriented language because every value is an object. Hence there are no primitives in Scala (unlike Java which has ,e.g., `int`s).



There are eight basic types in Scala: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, `Boolean`.
Every basic Scala type inherits from `AnyVal`. On the other side, `AnyRef` is an alias for `java.lang.Object`. Lastly, both `AnyVal` and `AnyRef` inherits from `Any`.

Numeric types :

```
1 + 1   // 2
2 - 1   // 1
5 * 3   // 15
6 / 2   // 3
6 / 4   // 1
6.0 / 4 // 1.5
6 / 4.0 // 1.5
```

Booleans. Scala supports the Boolean values `true` an `false`, as well as the common Boolean operations.

```
!true         // false
!false        // true
true == false // false
10 > 5        // true
```

Strings Strings are surrounded by double quotes. Single quotes only used for chars. Triple double-quotes let strings span multiple rows and contain quotes.

```
"Scala␣strings␣are␣surrounded␣by␣double␣quotes"
'a' // A Scala Char
val html = """<form␣id="daform">
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<p>Press␣belo',␣Joe</p>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<input␣type="submit">
␣␣␣␣␣␣␣␣␣␣␣␣␣␣</form>"""
```

String interpolation can be used to embed values directly in string literals.

```
val name = "Bob"
println(s"Hello␣$name!") // Hello Bob!
```

The `s` before the string is a string interpolator. Out of the box Scala provides three string interpolators, but we can create our own as well.

1. The **`s` interpolator** allows variables to be inserted into a string, as shown above. Arbitrary expressions can also be inserted:

   ```
   println(s"1 + 1 = ${1 + 1}")
   ```

2. The **`f` interpolator** allows creation of formatted strings. All variable references should be followed by a format string:

   ```
   val height = 1.9d
   val name = "James"
   println(f"$name%s is $height%2.2f meters tall")  // James is 1.90 meters tall
   ```

   Allowed format strings are outlines in the Formatter javadoc: `https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail` The `f` interpolator is typesafe. If you try to pass a format string that only works for integers but pass a double, the compiler will issue an error.

3. The **`raw` interpolator** is like the `s` interpolator, except it performs no escaping of literals within the string.

   ```
   scala> raw"a\nb"
   res1: String = a\nb
   ```

Array. Instantiation using an array literal and accessing elements of an array works as follows (with type inference):

```
val a = Array(1, 2, 3, 5, 8, 13)
a(0)     // Int = 1
a(3)     // Int = 5
a(21)    // Throws an exception
```

We can also declare an array as follows

```
val a = new Array[Int](2)
a(0) = 5
a(1) = 2
```

Despite being declared as a `val` in this example, the `Array` object is mutable so we can change the value of indexes 0 and 1. The designation `val` just means `a` cannot reference a different array. The array itself may be modified in situ.
A **`List`** is like an array, but is immutable.

Map. Initialization is as follows:

```
val colours = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")
colours("red") // java.lang.String = FF0000
```

A `Map` is an immutable data structure. Adding an element means creating another `Map`.
Maps can have a default value:

```
val safeColours = safeColours.withDefaultValue("Unknown")
safeColours("green")   // java.lang.String = "Unknown"
```

Mutable maps exist in `scala.collection.mutable.Map`.

```
val states = scala.collection.mutable.Map("AL" -> "Alabama", "AK" -> "tobedefined")
states("AK") = "Alaska"
```

Sets are Iterables that contain no duplicate elements. We test for set membership with parentheses.

```
val s = Set(1, 3, 7)
s(0)      // Boolean = false
s(1)      // Boolean = true
```

Tuples are immutable and contain a fixed number of elements, each with a distinct type. Literals use parentheses.

```
val d = (3, 1, "five")
(colours, states)
```

Accessing elements of tuples is done with `._n`, where `n` is the 1-based index of the element.

```
d._1    // Int = 3
d._2    // Int = 1
```

Tuples can also be unpacked:

```
val (div, mod) = divideInts(10, 3)
div     // Int = 3
mod     // Int = 1
```

Flow control

- **If-else**

  ```
  if (condition1) {

  } else if (condition2) {

  } else {

  }
  ```

  It is also an expression.

  ```
  def max(x: Int, y: Int) = if (x > y) x else y
  ```

- Basic for loop:

  ```
  // a goes from 0 to 10 inclusive
  for (a <- 0 to 10) {
    println(a)
  }
  // a goes from 0 to 10 exclusive
  for (a <- 0 until 10) {
  ```

```
  println(a)
}
// Looping over two elements
for (a <- 0 until 2; b <- 0 to 2) {
  println((a,b))
}
```

- Looping over iterable (with optional condition):

```
for (elem <- list if elem % 2 == 0) {

}
```

- For comprehension functions as a generator

```
val sub = for (elem <- list if elem % 2 == 0) yield elem
```

Exceptions can be handled in two ways.

```
try {
  val n = new FileReader("input.txt").read()
  println(s"Success: $n")
} catch {
  case e: Exception =>
    e.printStackTrace
}
```

Or using `Try`.

```
val tried: Try[Int] = Try(new FileReader("notes.md")).map(f => f.read())

tried match {
  case Success(n) => println(s"Success: $n")
  case Failure(e) => e.printStackTrace
}
```

Methods. A method is a function that is a member of a class, trait or object. A basic method example:

```
def add(x: Int, y: Int): Int = {
  x + y
}
```

Parameters are immutable. The `return` keyword is optional. The method will automatically return the last expression. `return` exits the current method, not the current block (as opposed to Java).
The return type is optional. The Scala compiler is also able to infer it.
Methods without output can be written two ways

```
def printSomething(s: String) = {
  println(s)
}
```

```
def printSomething(s: String): Unit = {
  println(s)
}
```

Multiple outputs can be returned using tuples.

```
def increment(x: Int, y: Int): (Int, Int) = {
  (x + 1, y + 1)
}
```

A method without arguments can be called without parentheses. Best practice dictates that this syntax is to be used if the method does not have any side-effects. (See also the `Dog` class below).
The last parameter can be allowed to take any number of arguments, using an asterisk

```
def variablesArguments(args: Int*): Int = {
  var n = 0
  for (arg <- args) {
    n += arg
  }
  n
}
```

We can also give parameters default values. The default parameter may be used by providing _ as an argument (or using named arguments).

```
def default(x: Int = 1, y: Int): Int = {
  x * y
}
default(_, 3)  // Int = 3
default(y = 3) // Int = 3
```

Methods may also be nested inside other methods.

Functions. Functions are first-class in Scala. This means methods can take a function as a parameter.

```
def foo(i: Int, f: Int => Int): Int = {
  f(i)
}
```

Function literals defined as follows

```
val increment: Int => Int = (x: Int) => x + 1
val divideInts: (Int, Int) => (Int, Int) = (x: Int, y: Int) => (x / y, x % y)
\\ Or, using type inference:
val divideInts = (x: Int, y: Int) => (x / y, x % y)
```

By not assigning a function to a variable, we get an anonymous function.
Scala supports closures.
Partial functions can be implemented with the following syntax

```
def speed(distance: Float, time: Float): Float = {
  distance / time
}
val partialSpeed: Float => Float = speed(5, _)
```

Classes, objects and traits

Classes Example of a class:

```
class Dog(br: String) {
  var breed: String = br
  def bark = "Woof, woof!"

  private def sleep(hours: Int) =
    println(s"I'm sleeping for $hours hours")
}

val mydog = new Dog("greyhound")
println(mydog.breed) // => "greyhound"
println(mydog.bark) // => "Woof, woof!"
```

Values and methods are assumed public. Values declared with `val` cannot be changed. `protected` (members are only accessible from sub-classes) and `private` (members are only accessible from the current class/object) keywords are also available. We can also make a method private only outside a package.
Scala supports two types of constructors:

1. The **primary constructor** is anything defined in the body of the class except method declarations. The parameter list comes after the class name and may contain default values. It may be omitted. Only a primary constructor is allowed to invoke a superclass constructor. A primary constructor may be made private by using the `private` keyword between the class name and the constructor parameter-list.

2. **Auxiliary contructors** are like methods with the name `this`. They must call a previously defined constructor, i.e. a primary or auxiliary constructor that lexically precedes it. The first statement of the auxiliary constructor must contain the constructor call using `this`.

   ```
   class GFG( Aname: String, Cname: String)
   {
       var no: Int = 0;;
       def display()
       {
           println("Author name: " + Aname);
           println("Chapter name: " + Cname);
           println("Total number of articles: " + no);

       }

       // Auxiliary Constructor
       def this(Aname: String, Cname: String, no:Int)
       {
           // Invoking primary constructor
           this(Aname, Cname)
           this.no=no
       }
   }
   ```

Abstract methods are methods with no body. A class with abstract methods must be declared `abstract`.

```
abstract class Dog(br: String) {
  var breed: String = br
  def chaseAfter(what: String): String
}
```

Objects The `object` keyword creates a type and a singleton instance of it. Objects and classes can have the same name.

```
object Dog {
  def allKnownBreeds = List("pitbull", "shepherd", "retriever")
  def createDog(breed: String) = new Dog(breed)
}
```

Case classes Case classes are like classes, but are primarily used to create data containers. They are immutable. Classes proper tend to focus on objects oriented concept, such as encapsulation, polymorphism, and behavior. The values tend to be private and only the methods exposed.

```
case class Person(name: String, phoneNumber: String)

val george = Person("George", "1234")
val kate = Person("Kate", "4567")
```

Cases classes don't need `new`.
Case classes have extra functionality built in, such as

- getters

  ```
  george.phoneNumber  // => "1234"
  ```

- per field equality (no need to override .equals)

  ```
  Person("George", "1234") == Person("Kate", "1236")  // => false
  ```

- easy way to copy

  ```
  val otherGeorge = george.copy(phoneNumber = "9876")
  ```

Traits Similar to Java interfaces, traits define an object type and method signatures. Scala allows partial implementation of those methods. Constructor parameters are not allowed. Traits can inherit from other traits or classes without parameters. A trait method can also have a default implementation.

```
trait Dog {
    def breed: String
    def color: String
    def bark: Boolean = true
    def bite: Boolean
}
```

```
class SaintBernard extends Dog {
    val breed = "Saint␣Bernard"
    val color = "brown"
    def bite = false
}
```

A trait can also be used as a mixin. The class "extends" the first trait, but the keyword `with` can add additional traits.

```
trait Bark {
    def bark: String = "Woof"
}
trait Dog {
    def breed: String
    def color: String
}
class SaintBernard extends Dog with Bark {
    val breed = "Saint␣Bernard"
    val color = "brown"
}
```

Generics TODO

Pattern matching Like a switch statement, but much more powerful.

```
def matchEverything(obj: Any): String = obj match {
  // You can match values:
  case "Hello␣world" => "Got␣the␣string␣Hello␣world"

  // You can match by type:
  case x: Double => "Got␣a␣Double:␣" + x

  // You can specify conditions:
  case x: Int if x > 10000 => "Got␣a␣pretty␣big␣number!"

  // You can match case classes as before:
  case Person(name, number) => s"Got␣contact␣info␣for␣$name!"

  // You can match regular expressions:
  case email(name, domain) => s"Got␣email␣address␣$name@$domain"

  // You can match tuples:
  case (a: Int, b: Double, c: String) => s"Got␣a␣tuple:␣$a,␣$b,␣$c"

  // You can match data structures:
  case List(1, b, c) => s"Got␣a␣list␣with␣three␣elements␣and␣starts␣with␣1:␣1,␣$b,␣$c'

  // You can nest patterns:
  case List(List((1, 2, "YAY"))) => "Got␣a␣list␣of␣list␣of␣tuple"
```

```
  // Match any case (default) if all previous haven't matched
  case _ => "Got unknown object"
}
```

Currying and implicits TODO

Concurrency TODO

5.5.6 Eiffel

Design by contract.

5.6 Interpreted languages

These are general-purpose programming languages that typically support multiple paradigms.

5.6.1 Python

special * syntax
PEP 3107 – Function Annotations PEP 484 – Type Hints

5.6.1.1 Implementations

CPython, PyPy.

5.6.2 Javascript

5.6.3 Lua

5.6.4 Perl

5.6.5 R

History, purpose and setup

Syntax

- In the interpreter you can get help about *word* using `?word`.
- **Comments**

  ```
  # Single line comments begin with a '#'
  # There are no multiline comments.
  ```

  item **Variables** can be assigned in different ways:

  ```
  x = 5     # this is possible
  y <- "1"  # this is preferred
  TRUE -> z # this works but is weird
  ```

Data and datatypes You can find out the type of any expression using `class()`:

```
class(5) # "numeric"
```

Basic types :

- **Numeric** is a double-precision floating-point number.

```
# Unless otherwise specified all numbers are assumed to be numerics
class(5)    # "numeric"
class(12.2) # "numeric"
# You can have infinitely large or small numbers
class(Inf)  # "numeric"
class(-Inf) # "numeric"
# You can also use scientific notation
5e4 # 50000
6.02e23 # Avogadro's number
1.6e-35 # Planck length
```

Illegal arithmetic yields a value `NaN` ("not-a-number"):

```
0 / 0 # NaN
class(NaN) # "numeric"
```

- **Integers** Long-storage integers are written with L.

```
5L # 5
class(5L) # "integer"
```

- **Character** There is no difference between strings and characters in R. To write a string literal both single and double quotes can be used.

```
class("Horatio")  # "character"
class('Horatio')  # "character"
class('H')        # "character"
```

- **Logical** Booleans are logical. Missing data (`NA`) is as well.

```
class(TRUE)     # "logical"
class(FALSE)    # "logical"
class(NA)       # "logical"
```

- **Factor** The factor class is for categorical data. Factors can be ordered (like childrens' grade levels) or unordered (like gender).
- **NULL** is NULL. It can be used to blank out a vector.

Data structures

- **Vectors** are created with the function `c()`.

```
c(1, 2, 3, 4)    # 1 2 3 4
vec <- c(8, 9, 10, 11)
vec              # 8 9 10 11
```

Every value is considered a vector of length 1. Conversely every vector has the datatype of its contents.

```
class(c(4L, 5L, 8L, 3L)) # "integer"
```

A vector can not contain data of different types, but it may always contain the logical value `NA`.

R indexes from 1. Slicing is also supported (bounds are inclusive).

```
vec[1]    # 8
vec[2:3]  # 9 10
```

Some other ways of creating vectors include

```
5:15                       # 5  6  7  8  9 10 11 12 13 14 15
seq(from=0, to=11, by=2)   # 0  2  4  6  8 10
```

- **Matrices** are two-dimensional vectors. All entries are of the same type. Unlike a vector, the class of a matrix is "matrix", no matter what's in it.

```
mat <- matrix(nrow = 4, ncol = 3, c(1,2,3,4,5,6))
mat
# =>
#       [,1] [,2] [,3]
# [1,]    1    5    3
# [2,]    2    6    4
# [3,]    3    1    5
# [4,]    4    2    6

class(mat) # "matrix"
```

Indexing for matrices:

```
# Ask for vector containing the first row
mat[1,]    # 1 5 3
# Ask for a specific cell
mat[3,2]   # 1
```

Matrices can be made by sticking vectors together, either as rows or as columns.

```
rbind(c(1,2,4,5), c(6,7,0,4))
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    4    5
# [2,]    6    7    0    4
cbind(1:4, c("dog", "cat", "bird", "dog"))
```

```
# =>
#      [,1] [,2]
# [1,] "1"  "dog"
# [2,] "2"  "cat"
# [3,] "3"  "bird"
# [4,] "4"  "dog"
```

- **Data frames** are two-dimensional structures that can contain different types. The class of a data frame is "data.frame".

```
students <- data.frame(c("Cedric","Fred","George","Cho","Draco","Ginny"),
                       c(3,2,2,1,0,-1),
                       c("H", "G", "G", "R", "S", "G"))
names(students) <- c("name", "year", "house") # name the columns
class(students)   # "data.frame"
students
# =>
#     name year house
# 1 Cedric    3     H
# 2   Fred    2     G
# 3 George    2     G
# 4    Cho    1     R
# 5  Draco    0     S
# 6  Ginny   -1     G
```

The `data.frame()` function converts character vectors to factor vectors by default; turn this off by setting `stringsAsFactors = FALSE` when you create the data frame.

Indexing data frames:

```
students$year       # 3  2  2  1  0 -1
students[,2]        # 3  2  2  1  0 -1
students[,"year"]   # 3  2  2  1  0 -1
```

A column can be dropped by assigning the `NULL` value to it.

```
students$house <- NULL
students
# =>
#     name year
# 1 Cedric    3
# 2   Fred    2
# 3 George    2
# 4    Cho    1
# 5  Draco    0
# 6  Ginny   -1
```

Rows can be dropped by subsetting:

```
students[students$house != "G",]
# =>
#     name year house
# 1 Cedric    3     H
```

```
# 4    Cho    1     R
# 5  Draco    0     S
```

- **Arrays** are n-dimensional structures that contain only one type.

```
array(c(c(c(2,300,4),c(8,9,0)),c(c(5,60,0),c(66,7,847))), dim=c(3,2,2))
# =>
# , , 1
#
#      [,1] [,2]
# [1,]    2    8
# [2,]  300    9
# [3,]    4    0
#
# , , 2
#
#      [,1] [,2]
# [1,]    5   66
# [2,]   60    7
# [3,]    0  847
```

- **Lists** are like dictionaries in Python. May be multi-dimensional. Possibly ragged. May contain different types.

```
list1 <- list(time = 1:40)
list1$price = c(rnorm(40,.5*list1$time,4))
```

List indexing can be done in several ways. The following are equivalent in this case:

```
list1$time
list1[["time"]]
list1[[1]]
```

Lists are not very efficient.

```
as.character(c(6, 8)) # "6" "8"
as.logical(c(1,0,1,1)) # TRUE FALSE  TRUE  TRUE
as.numeric("Bilbo")
# =>
# [1] NA
# Warning message:
# NAs introduced by coercion
```

If you put elements of different types into a vector, weird coercions happen:

```
c(TRUE, 4) # 1 4
c("dog", TRUE, 4) # "dog"  "TRUE" "4"
```

Basic operations and arithmetic

```
TRUE == FALSE   # FALSE
FALSE != TRUE   # TRUE
```

Numbers Doing arithmetic on a mix of integers and numerics returns a numeric. Arithmetic on integers returns integers (except with division).

```
10L + 66L # 76
53.2 - 4  # 49.2
2.0 * 2L  # 4
3L / 4    # 0.75    # dividing always returns numeric
4 ^ 2     # 16
4 %% 3.1  # 0.9     # modulo
4 %% 3.1  # 0.9     # modulo
```

```
# OR
TRUE | FALSE    # TRUE
TRUE | NA       # TRUE
FALSE | NA      # NA
NA | NA         # NA

# AND
TRUE & FALSE    # FALSE
TRUE & NA       # NA
FALSE & NA      # FALSE
NA & NA         # NA
```

Vectors

- Arithmetic with vectors of the same length pairs up the elements

  ```
  c(1,2,3) + c(1,2,3) # 2 4 6
  ```

- Arithmetic with scalars is applied element-wise to the vector

  ```
  (4 * c(1,2,3) - 2)  # 2 6 10
  ```

- Arithmetic with vectors of different length can only be done if the length of the larger vector is an integer multiple of the length of the smaller. The smaller vector is then repeated enough times to fill the larger. This is a generalisation of the above two behaviours. Usually it is better practice and easier to read if lengths are matched.

  ```
  c(1,2,3,1,2,3) * c(1,2)          # 1 4 3 2 2 6
  c(1,2,3,1,2,3) * c(1,2,1,2,1,2)  # 1 4 3 2 2 6

  c('Z', 'o', 'r', 'r', 'o') == "Zorro"  # FALSE FALSE FALSE FALSE FALSE
  c('Z', 'o', 'r', 'r', 'o') == "Z"      # TRUE FALSE FALSE FALSE FALSE
  # (Remember every value is treated as a vector of length 1)
  ```

Matrices Matrices can be transposed and multiplied.

```
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6
mat %*% t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]   17   22   27
# [2,]   22   29   36
# [3,]   27   36   45
```

Flow control and loops

- **If/else**

```
if (4 > 3) {
    print("4 is greater than 3")
} else {
    print("4 is not greater than 3")
}
```

- **For loops**

```
for (i in 1:4) {
  print(i)
}
```

- **While loops**

```
a <- 10
while (a > 4) {
    cat(a, "...", sep = "")
    a <- a - 1
})
```

Loops run slowly in R. It is generally much better to do operations on entire vectors or use `apply()`-type functions.

Environments

```
jiggle <- function(x) {
    x = x + rnorm(1, sd=.1) #add in a bit of (controlled) noise
    return(x)
}
# Called like any other R function:
jiggle(5)
```

Built-in functionality

- **Constants**

letters

month.abb

- **Numeric functions**

round()

log()

max()

- **Logical functions**

isTRUE()

- **Functions on strings**

substr()

gsub()

- **Functions on vectors**

length()

sort()

which() Return indices of elements that match.

any() Return true if any of the elements match.

max()

min()

sum()

head() Look at top of dataset.

tail() Look at bottom of dataset.

Descriptive statistics

data() Browse pre-loaded data sets

data(rivers) Load dataset "Lengths of Major North American Rivers" as a numeric vector `rivers`.

mean()

var()

`sd()`

`summary(rivers)` Summary statistics: minimum, 1st quartile, median, mean, 3rd quartile, maximum.

- **Functions on data frames**

`dim()`

`nrow()`

`ncol()`

- **Data visualisation**

Stem-and-leaf

Histogram

Plot

Packages The `data.table` package provides functionality a lot like the data frames.

```
# An augmented version of the data.frame structure is the data.table
# If you're working with huge or panel data, or need to merge a few data
# sets, data.table can be a good choice. Here's a whirlwind tour:
install.packages("data.table") # download the package from CRAN
require(data.table) # load it
students <- as.data.table(students)
students # note the slightly different print-out
# =>
#      name year house
# 1: Cedric    3     H
# 2:   Fred    2     G
# 3: George    2     G
# 4:    Cho    1     R
# 5:  Draco    0     S
# 6:  Ginny   -1     G
students[name=="Ginny"] # get rows with name == "Ginny"
# =>
#     name year house
# 1: Ginny   -1     G
students[year==2] # get rows with year == 2
# =>
#      name year house
# 1:   Fred    2     G
# 2: George    2     G
# data.table makes merging two data sets easy
# let's make another data.table to merge with students
founders <- data.table(house=c("G","H","R","S"),
                       founder=c("Godric","Helga","Rowena","Salazar"))
founders
# =>
#    house founder
# 1:     G  Godric
```

```
# 2:     H   Helga
# 3:     R  Rowena
# 4:     S Salazar
setkey(students, house)
setkey(founders, house)
students <- founders[students] # merge the two data sets by matching "house"
setnames(students, c("house","houseFounderName","studentName","year"))
students[,order(c("name","year","house","houseFounderName")), with=F]
# =>
#    studentName year house houseFounderName
# 1:        Fred    2     G           Godric
# 2:      George    2     G           Godric
# 3:       Ginny   -1     G           Godric
# 4:      Cedric    3     H            Helga
# 5:         Cho    1     R           Rowena
# 6:       Draco    0     S          Salazar

# data.table makes summary tables easy
students[,sum(year),by=house]
# =>
#    house V1
# 1:     G  3
# 2:     H  3
# 3:     R  1
# 4:     S  0
```

5.7 Functional languages

5.7.1 Functional programming paradigm

pure state,

5.7.2 LISP

5.7.3 Scheme

Continuations?

5.7.4 ML

5.7.5 Haskell

History.

Concepts. Haskell has lazy evaluation; it only evaluates things when it needs to.

Basic setup. Haskell can be run in a REPL (Read-Eval-Print Loop). The REPL can be started with the command `ghci`.
In the REPL, new values are created with `let`.

```
let foo = 5
```

Type can be inspected using `:t`.

```
> :t foo
foo :: Integer
```

Additional information on any identifier is given by `:i`:

```
> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in 'GHC.Num'
infixl 6 +
```

Syntactic elements. Comments:

```
-- Single line comments start with two dashes.
{- Multiline comments can be enclosed
in a block like this.
-}
```

Lines end with a newline character.

Primitive data types and operators.

Numbers. Math is a you expect. Division is floating point by default. Integer division done using `` `div` ``.

```
1 + 1 -- 2
8 - 1 -- 7
10 * 2 -- 20
35 / 4 -- 8.75
35 `div` 4 -- 8
```

Booleans. The primitives `True` and `False` are capitalised. Operations:

```
not True -- False
not False -- True
1 == 1 -- True
1 /= 1 -- False
1 < 10 -- True
```

Here `not` is actually a function.

Strings and characters. Strings are lists of characters.

```
"This␣is␣a␣string."
'a' -- character
'You cant use single quotes for strings.' -- error!

['H', 'e', 'l', 'l', 'o'] -- "Hello"
"This␣is␣a␣string" !! 0 -- 'T'
```

Function identifiers do not need to contain letters.

```
(//) a b = a `div` b
35 // 4 -- 8
```

Lists and tuples.

Lists. Every element in a list must have the same type. Ranges can be used and are versatile.

```
[1, 2, 3, 4, 5]
[1..5]              -- [1, 2, 3, 4, 5]
['A'..'F']          -- "ABCDEF"
[0,2..10] -- [0, 2, 4, 6, 8, 10]
[5..1] -- [] (Haskell defaults to incrementing)
[5,4..1] -- [5, 4, 3, 2, 1]
```

Indexing is zero-based and done using `!!`.

```
[1..10] !! 3 -- 4
```

Thanks to lazy evaluation the following is possible:

```
[1..] -- a list of all the natural numbers
[1..] !! 999 -- 1000
```

List operations:

```
-- joining two lists
[1..5] ++ [6..10]

-- adding to the head of a list
0:[1..5] -- [0, 1, 2, 3, 4, 5]

head [1..5] -- 1
tail [1..5] -- [2, 3, 4, 5]
init [1..5] -- [1, 2, 3, 4]
last [1..5] -- 5
```

List comprehension:

```
[x*2 | x <- [1..5]] -- [2, 4, 6, 8, 10]
[x*2 | x <- [1..5], x*2 > 4] -- [6, 8, 10]
```

Tuples. Every element in a tuple can be a different type, but a tuple has a fixed length. Tuple literals are written with parentheses.

```
("haskell", 1)

-- accessing elements of a pair (i.e. a tuple of length 2)
fst ("haskell", 1) -- "haskell"
snd ("haskell", 1) -- 1

-- pair element accessing does not work on n-tuples (i.e. triple, quadruple, etc)
snd ("snd", "can't␣touch␣this", "da␣na␣na␣na") -- error! see function below
```

Functions. Declaring and calling functions.

```
add a b = a + b
add 1 2 -- 3
```

Haskell also supports infix notation.

```
1 `add` 2 -- 3
```

Branching can be achieved with guards.

```
fib x
  | x < 2 = 1
  | otherwise = fib (x - 1) + fib (x - 2)
```

```
fib 1 = 1
fib 2 = 2
fib x = fib (x - 1) + fib (x - 2)
```

Pattern matching.

- On tuples:

  ```
  sndOfTriple (_, y, _) = y
  ```

- On lists:

  ```
  myMap func [] = []
  -- x is the first element of the list.
  -- xs is the rest of the list.
  myMap func (x:xs) = func x:(myMap func xs)
  ```

Anonymous functions. Anonymous functions are created with a backslash followed by all the arguments.

```
myMap (\x -> x + 2) [1..5] -- [3, 4, 5, 6, 7]
```

```
add a b = a + b
foo = add 10 -- foo is now a function that takes a number and adds 10 to it
foo 5 -- 15
-- Another way to write the same thing
foo = (10+)
foo 5 -- 15
```

Function composition. Function composition is achieved with the `.` operator.

```
foo = (4*) . (10+)
foo 5 -- 60 because 4*(10+5) = 60
```

Operator precedence. The `$` operator applies a function to a given parameter. It is low priority and is right-associative. The expression on its right is applied as a parameter to the function on its left.

Built in functions Another `map` example.

```
map (*2) [1..5] -- [2, 4, 6, 8, 10]
```

foldr, foldl

Type signatures and data types. Haskell has a very strong type system, and every valid expression has a type.
Some basic types:

```
5 :: Integer
"hello" :: String
True :: Bool
```

When you define a value, it's good practice to write its type above it:

```
doubleInt :: Integer -> Integer
doubleInt x = x * 2
```

Custom types can be defined.

```
data Color = Red | Blue | Green
```

The type is `Color` and its possible values are `Red`, `Blue` and `Green`. Data types can have parameters as well.

```
data Maybe a = Nothing | Just a
-- These are all of type Maybe
Just "hello"    -- of type `Maybe String`
Just 1          -- of type `Maybe Int`
Nothing         -- of type `Maybe a` for any `a`
```

Flow control.

```
haskell = if 1 == 1 then "awesome" else "awful" -- haskell = "awesome"
```

Multiline `if`. Indentation is important.

```
haskell = if 1 == 1
            then "awesome"
            else "awful"
```

```
case args of
  "help" -> printHelp
  "start" -> startProgram
  _ -> putStrLn "bad␣args"
```

Recursion. Haskell does not have any loops, but we can make them using the `map` function.

```
for array func = map func array
for [0..5] $ \i -> show i -- Using the for loop.
```

Monads. TODO

I/O. TODO

5.8 Logic programming languages

5.8.1 Planner

Inspired smalltalk and prolog

5.8.2 Prolog

History First specified in 1972. SWI-Prolog offers a comprehensive free Prolog environment.

Concepts

- A Prolog database consists of known instances of relations, called facts.
- New relations can be constructed from old ones. These are expressed as rules.
- Queries are expressions containing one or more variables.
- Unification is used to determine whether a query has a valid substitution consistent with the rules and facts in the database.
- Clauses are sets of statements.
- A subprogram (called a predicate) represents a state of the world.
- A command (called a goal) tells Prolog to make that state of the world true, if possible.

Basic setup Code entered in interactive mode and code in a file is treated differently. Facts and rules should be put in a file.
The interactive prompt is `?-` and lines end with a period.

Syntactic elements Comments:

```
% This is a comment
```

Facts:

```
magicNumber(7).
magicNumber(9).
magicNumber(42).
```

which we can query:

```
?- magicNumber(7).                    % True
?- magicNumber(8).                    % False
?- magicNumber(9).                    % True
```

Multiple operations can be chained using commas.

Unification We request unification by passing an undefined variable:

```
?- magicNumber(Presto).                % Presto = 7 ;
                                       % Presto = 9 ;
                                       % Presto = 42.
```

The equals sign represents unification. We have three cases.

1. If both sides are bound (i.e., defined), Prolog checks equality.
2. If one side is free (i.e., undefined), Prolog tries to assign the variable to match the other side.
3. If both sides are free, the assignment is remembered.

Attempted unification can have three outcomes. It can

1. Succeed (return True) without changing anything, because an equality-style unification was true;
2. Succeed (return True) and bind one or more variables in the process; or
3. Fail (return False) because an equality-style unification was false (failure can never bind variables).

The equals sign can not do arithmetic (as Prolog cannot solve equations out of the box). The `is` operator does allow arithmetic, but right side must always be bound.

```
?- X = 3+2.              % X = 3+2 - unification can't do arithmetic
?- X is 3+2.             % X = 5 - "is" does arithmetic.
?- 5 = X+2.              % This is why = can't do arithmetic -
                         % because Prolog can't solve equations
?- 5 is X+2.             % Error. Unlike =, the right hand side of IS
                         % must always be bound, thus guaranteeing
                         % no attempt to solve an equation.
```

We can however reverse addition if we try to unify with the `plus` predicate.

```
?- plus(1, 2, 3).                      % True
?- plus(1, 2, X).                      % X = 3 because 1+2 = X.
?- plus(1, X, 3).                      % X = 2 because 1+X = 3.
?- plus(X, 2, 3).                      % X = 1 because X+2 = 3.
?- plus(X, 5, Y).                      % Error - infinite solutions
```

TODO: more

5.9 Numerical computing

5.9.1 MATLAB

5.9.2 GNU Octave

5.9.3 Maxima

5.9.4 Julia

5.10 Shell scripting

5.10.1 Bash

5.10.2 DOS

5.11 System and circuit design

5.11.1 SPICE

5.11.2 Verilog

5.11.3 Chisel

5.11.4 FIRRTL

5.11.5 LabVIEW and G

5.12 Database querying

5.12.1 SQL

5.13 Data-interchange

5.13.1 XML

5.13.2 JSON

Part IV

Computer systems

Chapter 1

Process management and coordination

Chapter 2

Memory and storage

bit, byte, word, kilo / kibi

Chapter 3

UI / UX

Chapter 4

Networking

TCP / IP etc. ATM
Hosting: firebase

Chapter 5

Computer security

A chain is only as strong as its weakest link.

5.1 Cryptography

Shamir's Secret Sharing

5.1.1 Stenography

Chapter 6

Distributed systems

Chapter 7

Operating systems

POSIX MBR / UEFI ... Encoding (Unicode / UTF-8) ... Console: Bourne, C shell, Bourne-Again, Korn shell, Bash, busybox, DOS, Cygwin, MinGW
Also mobile and game console

Part V

Advanced algorithms

Chapter 1

Artificial intelligence

Chapter 2

Machine learning

Chapter 3

Simulation

3.1 Overview and problem statement

3.1.1 Time and length scales

Multiscale modeling: continuum, meso, nano, QM

3.1.2 Atomic scale

Trajectories over potential energy hypersurfaces; MD uses thermal energy to move smoothly over surface, MC moves randomly with probability $\exp(-\Delta U/k_B T)$.
Equilibrium properties ore calculated as mean values over M sampled configurations (assuming the configuration space has been sampled according to the correct distribution)

$$\bar{f} = \frac{1}{M} \sum_{\alpha=1}^{M} f_\alpha$$

Law of large numbers:

$$\mathbb{E}[f] = \lim_{M \to \infty} \bar{f}$$

Complex structure	**Length scale**	**Time scale**	**Mechanics**
Complex structure	10^3m	10^6s	structural mechanics
Simple structure	10^1m	10^3s	fracture mechanics
Component	10^{-1}m	10^0s	Continuum mechanics
Grain microstructure	10^{-3}m	10^{-3}s	crystal plasticity
Dislocation microstructure	10^{-5}m	10^{-6}s	micro-mechanics
Single dislocation	10^{-7}m	10^{-9}s	dislocation dynamics
Atomic	10^{-9}m	10^{-12}s	molecular dynamics
Electron orbitals	10^{-11}m	10^{-15}s	quantum mechanics

Table 3.1: The important unit structure is indicated for each scale, along with approximate length and time scales, as well as the approach used to simulate the material's behaviour.

3.1.3 Thermodynamic quantities

- **Internal energy** Sum of kinetic and potential energy

$$U = \bar{K} + \bar{V} \qquad \bar{K} = \sum_{i=1}^{N} \frac{\bar{p}_i^2}{2m_i}$$

- **Temperature** calculated using the equipartition theorem

$$\bar{K} = \frac{3}{2} N k_B T \qquad \rightarrow \qquad T = \frac{2}{3Nk_B}\bar{K} = \frac{2}{(3N_a - N_c)k_B}\bar{K}$$

  where N_a is number of atoms and N_c is the total number of independent constraints.

- **Pressure** is calculated using the expression obtained from the virial theorem

$$p = \frac{N}{V} k_B T + \frac{1}{3V} \sum_{i=1}^{N} \sum_{j>i}^{N} \mathbf{f}_{ij} \bar{\cdot} \mathbf{r}_{ij}$$

- **Free energies** and **entropy** cannot be calculated directly in general because these are properties of the whole ensemble which require counting all of the ensemble states. There are however some tricks to calculate free energy differences.

3.1.4 Fluctuations

Trajectory length must be $>$ decay time of ACF
Time separation must be $\ll$ decay time of ACF.

3.1.5 Block averaging

3.2 Monte Carlo simulations

3.2.1 Calculating area under a curve

3.2.2 Importance sampling

3.2.3 Boundary conditions

3.2.4 Markov chain Monte Carlo

A Markov chain is a stochastic model describing a sequence of events where the probability of the next event occurring only depends on the previous event. In other words we have a chain

$$X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_J \rightarrow X_{J+1} \rightarrow \ldots$$

and the Markov model is completely determined by the transition probabilities $W(X_J \rightarrow X_{J'})$.

3.2.5 Metropolis algorithm

A Monte Carlo algorithm can be used to compute (approximations of) ensemble averages of quantities in statistical mechanics. This is done by sampling the ensemble, calculating the quantity for each sample and then averaging the results.
The tricky part is being able to accurately sample the ensemble. The Metropolis algorithm provides a way to do that with Markov chains. Not all Markov chains sample a well-defined, stationary distribution. A sufficient but not necessary condition for the existence of a stationary distribution is detailed balance.
Detailed balance is essentially microscopic reversibility. For every pair of states J, J' the probability of being in state J and transitioning to J' is the same as being in state J' and transitioning to J.

$$W_{J' \leftarrow J} P_J^{\text{eq}} = W_{J \leftarrow J'} P_{J'}^{\text{eq}}$$

It is exactly this stationary distribution P_J^{eq} that we want to reflect the ensemble. The equality can be rewritten as

$$\frac{P_J^{\text{eq}}}{P_{J'}^{\text{eq}}} = \frac{W_{J \leftarrow J'}}{W_{J' \leftarrow J}}$$

Next a new and interesting way to generate the Markov chain is introduced. Each state is generated from the previous one using a two-step process:

1. Propose a new state J'. The likelihood of a new state J' being proposed is given by $\alpha_{J' \leftarrow J}$.
2. Decide whether to accept or reject the new state. The new state is accepted with probability $W^{\text{acc}}_{J' \leftarrow J}$.

The transition probabilities are then given by

$$W_{J' \leftarrow J} = \alpha_{J' \leftarrow J} W^{\text{acc}}_{J' \leftarrow J}.$$

Further imposing detailed balance gives

$$\frac{P_J^{\text{eq}}}{P_{J'}^{\text{eq}}} = \frac{W_{J \leftarrow J'}}{W_{J' \leftarrow J}} = \frac{\alpha_{J \leftarrow J'} W^{\text{acc}}_{J \leftarrow J'}}{\alpha_{J \leftarrow J'} W^{\text{acc}}_{J' \leftarrow J}}$$

The beauty of the Metropolis algorithm is that if the underlying stochastic matrix α is symmetric ($\alpha_{J \leftarrow J'} = \alpha_{J' \leftarrow J}$), then this further simplifies to

$$\frac{P_J^{\text{eq}}}{P_{J'}^{\text{eq}}} = \frac{W^{\text{acc}}_{J \leftarrow J'}}{W^{\text{acc}}_{J' \leftarrow J}}$$

This means we can choose any α we like (so long as it's symmetric) and we can make sure we are sampling the ensemble correctly by using good rules for accepting new moves.

3.2.5.1 Sampling the canonical ensemble

If we are trying to sample the canonical (NVT) ensemble, we need the stationary distribution the be the Boltzmann distribution.

$$\frac{P_J^{\text{eq}}}{P_{J'}^{\text{eq}}} = \exp\left[-\frac{V_{J'} - V_J}{k_B T}\right]$$

This still leaves some choice as to the exact form of $W^{\text{acc}}_{J' \leftarrow J}$. The Metropolis choice is

$$W^{\text{acc}}_{J' \leftarrow J} = \begin{cases} \exp\left[-\frac{V_{J'} - V_J}{k_B T}\right] & (V_{J'} > V_J) \\ 1 & (V_{J'} \leq V_J) \end{cases}$$

in other words

$$W^{\text{acc}}_{J' \leftarrow J} = \min\left[1,\ \exp\left(-(V_{J'} - V_J)/k_B T\right)\right]$$

TODO write out algorithm

Part VI

Applications

Chapter 1

Computer graphics

1.1 Graphics formats

svg, jpeg, png

1.2 2D graphics creation

Inkscape, GIMP,

1.3 3D graphics creation

Blender

1.3.1 Rendering

1.3.1.1 Ray-tracing

1.4 Computer vision

OpenCV
face-recognition

Chapter 2

The data analysis workflow

Chapter 3

Web

Chapter 4

Games and game engines

Chapter 5

Publishing

tex
latex
tikz

Chapter 6

Audio

alsa, jack, pulseaudio

Part VII

Reference

Appendix A

Bibliography

Software Languages: Syntax, Semantics, and Metaprogramming by Ralf Lämmel. Lämmel, Ralf
The Art of Computer Programming, D. Knuth
Introduction to Algorithms, third edition
A categorical manifesto
https://github.com/jozefg/learn-tt
Spartan type theory
Homotopy Type Theory: Univalent Foundations of Mathematics https://homotopytypetheory.org/book/
Gang of four design patterns
Introduction to Univalent Foundations of Mathematics with Agda by Martín Escardó.
Types and programming languages, Benjamin C. Pierce
An introduction to univalent foundations for mathematicians, DANIEL R. GRAYSON
Proofs and Types, Jean-Yves Girard
Bengt Nordström, Kent Petersson, and Jan M. Smith: Programming in Martin-Löf's Type Theory
Advanced Compiler Design and Implementation Third edition Programming Languages: design and implementation. Terrence W. Pratt and Marvin V. Zelkowitz
Donald Knuth - Structured programming with go to statements https://web.archive.org/web/20190421081706/http://www.cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGo
pdf
Dijkstra, E. W. (March 1968). "Letters to the editor: go to statement considered harmful". Communications of the ACM. 11 (3): 147–148. doi:10.1145/362929.362947.
Martin Fowler: UML Distilled, third edition. Executable UML: A Foundation for Model-Driven Architecture, Mellor, Balcer Object Oriented Systems development using the unified modeling language, Bahrami
Pattern recognition, an algorithmic approach, Natashima Unifying theories of programming, Hoare
Dijkstra, Edsger W. "EWD472: Guarded commands, non-determinacy and formal. derivation of programs" http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF
The definitive ANTLR 4 reference, Terence Parr
FORTRAN 77 tutorial (https://web.stanford.edu/class/me200c/tutorial_77/)
http://fortranwiki.org/fortran/show/HomePage
https://learnxinyminutes.com/

https://docs.scala-lang.org https://hackernoon.com/a-10-minute-introduction-to-scala-c
Programming in Haskell (2nd ed), Graham Hutton
ToRead: http://www.drdobbs.com/architecture-and-design/so-you-want-to-write-your-own-l
240165488?pgno=1 https://hackernoon.com/considerations-for-programming-language-design
- OO bad http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.
html https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53
https://www.yegor256.com/2016/08/15/what-is-wrong-object-oriented-programming.
html https://content.pivotal.io/blog/all-evidence-points-to-oop-being-bullshit
https://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey - HDL
http://www.myhdl.org/ http://pshdl.org/ https://hackage.haskell.org/package/
kansas-lava https://chisel.eecs.berkeley.edu/documentation.html https:
//chisel.eecs.berkeley.edu/chisel-dac2012.pdf https://github.com/freechipsproject/
firrtl/blob/master/spec/spec.pdf
https://insights.sigasi.com/opinion/jan/verilogs-major-flaw.html https:
//www.reddit.com/r/programming/comments/5avlpa/why_hardware_development_
is_hard_part_1_verilog/ https://www.viewpointusa.com/IE/ar/hdl-fpga-development-the-go
https://news.ycombinator.com/item?id=7565711 https://danluu.com/why-hardware-developme
- Go https://tour.golang.org/list https://blog.golang.org/go-slices-usage-and-internals
https://blog.golang.org/defer-panic-and-recover https://blog.golang.org/
go-concurrency-patterns-timing-out-and https://blog.golang.org/go-maps-in-action
https://blog.golang.org/gos-declaration-syntax https://www.quora.com/
Why-does-Go-seem-to-be-the-most-heavily-criticized-among-the-newer-programming-langua
https://github.com/ksimka/go-is-not-good http://nomad.uk.net/articles/
why-gos-design-is-a-disservice-to-intelligent-programmers.html https:
//news.ycombinator.com/item?id=9266184 https://bluxte.net/musings/2018/
04/10/go-good-bad-ugly/ https://talks.golang.org/2012/splash.article https:
//bravenewgeek.com/go-is-unapologetically-flawed-heres-why-we-use-it/
https://hackernoon.com/the-beauty-of-go-98057e3f0a7d https://corte.si/
posts/code/go/golang-practicaly-beats-purity/index.html - Scala https://
news.ycombinator.com/item?id=8420199 https://upon2020.com/blog/2017/05/
why-scala-is-not-for-you-nor-me/ - Generics
Head first design patterns; Freeman, Robson