# Application Generation

**Machine Code**

- Computers only understand machine code, consisting of just binary sequences split into instructions and data (opcodes and operands).

- For the first computers there was no choice other than to write in machine code.

- Would be error prone. Different processors had different **instruction sets** so the code was not portable. Programs need rewriting for each computer system.

**Assembler**

- Consists of mnemonics that are easier to remember than opcodes but which have a 1:1 relationship with the opcodes in an instruction set.

- An assembler converts assembly code into machine code.

- Essentially translates directly into binary word by word.

- Assembly isn’t very portable either.

**Compilers and Interpreters**

- A higher level language consists of more human readable code.

- Requires converting into machine code via either an interpreter or compiler.

**Interpreter**

- An interpreter takes each line of a high-level language program, converts it to machine code and runs it.

- This is useful when debugging as the program can start right away and stop when it encounters an error.

- The downside is that the program runs slower. The user has to wait for the translation of each line as well as the translation.

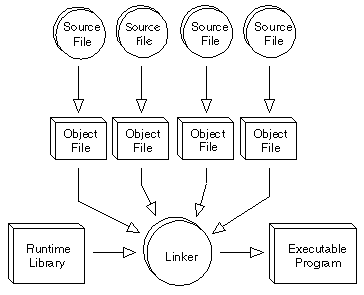
- The interpreter may have to translate the same line many times in a loop which is inefficient.

**Compiler**

- A compiler is a program that takes a program written in a high-level language and converts it to **object code.**

- The object code can then be distributed to anyone with a compatible system without the need for additional programs.

- Once the program has been compiled it can be run as often as needed faster than its interpreted equivalent.

- The object code is not human readable which strengthens anti piracy measures for commercial code.

- Compilation may take a while for large programs.

**Object Code**

- Often used interchangeably with machine code but object code is an intermediary step sometimes taken before pure machine code is produced.

- The object code contains place-holders where library code needs to go. Once a linker has been sued machine code that can be run is produced.

**How a Compiler Works**

- A compiler works by going through a sequence of stages, each moving closer to machine code. While the exact process varies between compilers. Most will include the stages:  
 - Lexical Analysis.

- Syntax Analysis.

- Code Generation.

- Optimisation.

**Lexical Analysis**

**Reserved Word**

A word that has special meaning in a programming language and as such cannot be used as a variable name. E.g: if, else, while and for.

- All comments and white-space are removed from the program.

- High-level code is turned into a series of **tokens** which represent **reserved words**, operators, variables and constants. Tokens are specific strings of characters.

**Syntax Analysis**

- The syntax of a language is the set of rules that govern its structure.

- Syntax analysis checks the code that has been written uses a valid syntax. Where code does not follow the rules of the language the compiler will generate a list of syntax errors to alert the programmer.

- Syntax analysis will produce an Abstract Syntax Tree (**AST**) that will represent the program. If the tokens do not fit into this structure then this means there is a syntax error.

**Code Generation**

- At this stage code is represented by an Abstract Syntax Tree. Code generation is when the compiler converts this into object code.

**Optimisation**

- Optimisation makes sure code runs as quickly as possible.

- There are a number of tricks the compiler can use to achieve this:

- Remove lines of code that have no effect on the program.

- It will look at instructions and see if they can be replaced by a more useful alternative.

- Optimisation happens during and after **code generation.**

**Libraries**

- Often code to perform complex tasks has already been written. This code can be reused by other programmers.

- It is best to do use libraries where possible. They are often designed to tackle a complex task such as graphics or cryptography which may otherwise be time consuming to write yourself.

- Programmers use libraries through an **API (Application Programming Interface)** a library may be written in one language and the have APIs designed to work in other languages.

**Linkers and Loaders**

- Once code has been generated and optimised, it is still not quite ready to be run. There is a good chance the code will rely on libraries.

- The job of a linker is to include this library code and all the compiled files into the final single executable program. Linkers can either be **static** or **dynamic** linkers (which are really loaders).

- When using static linking, all the library code needed is put directly into the program when it is compiled.

- This means programs can be bulky and a single computer could have a number of different programs each with their own separate copies of libraries wasting space.

- Dynamic linking circumvents this problem. Compiled versions of the library are stored and the Operating System links a program to them when it is run. A loader is part of the operating system and is responsible for loading a program into memory.