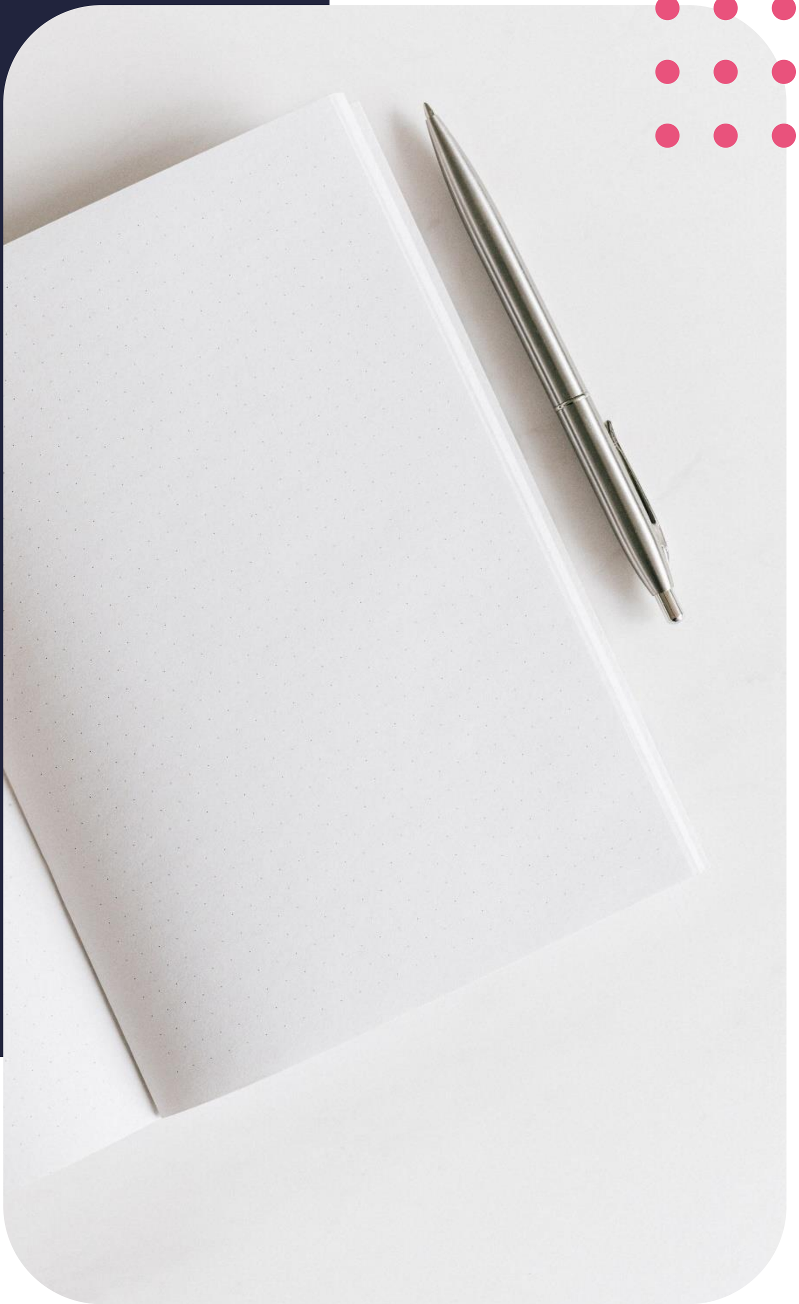
Programming languages

Contents



Lesson outcomes

By the end of this lesson, you should be able to:

- Examine programming language paradigms
- Introduce the factors to consider when choosing a programming language.
- Discuss the two major levels of programming languages
- Explain the compilation process

History of programming languages

History of programming language

The roots of programming languages extend over 200 years back to the Jacquard machine that was fitted to a loom to simplify the manufacture of fabrics with complex patterns. While it wasn't programming in the sense that we understand it today, it laid the foundation for telling machines what to do.

An entire century would go by before electrical computers, as we know them today, would come into existence.

Programming languages evolved along with computers because they are directly related to the type of computer they target. The first computer programmer was a female mathematician, Ada Lovelace, who set out the algorithm for Charles Baggage's Analytical with the goal of calculating Bernoulli numbers. Here's an interesting snippet – on the 15th October every year, Lovelace is acknowledged to highlight the often-overlooked contributions of women in the fields of Maths and Science.

It took a really long time for the next major development. When the first stored-program computers appeared in the late 1940s, the development of programming languages sprouted thanks to British computer scientist Alick Glennie. This was the first programming language known as Autocode, built for the Mark 1 computer.

We're going to look at compiling a bit later but, for now, let's continue with the evolution of programming language.

So, in those early days, programming was done using machine language. This was very tedious and often plagued with errors. The late 1940s brought a significant development: chunks of machine code were converted into words like ADD, SUB and MULT, which made programming a little easier. Another female, Kathleen Booth, can be credited with inventing assembly language. And, at the time, it was a real breath of fresh air for programmers because it meant that they didn't have to remember strings of binary code and calculate addresses.

Assembly language – which is low-level programming language – is still used today for direct hardware manipulation, such as in device drivers, low level embedded systems and real time systems. There are instances where it's simply more logical to use assembly language. And we will look at this a bit down the line.

During the 60s and 70s many new programming languages were invented, consolidated and elaborated. Languages like C evolved into C++.

The focus was on programming large scale systems using modules. Older programming languages were adapted to new contexts, such as object-oriented programming.

With advent of the Internet Age, which itself was a radical new platform for computer programming, the focus switched to programmer productivity, which sparked the development of scripting (or interpreted) languages.

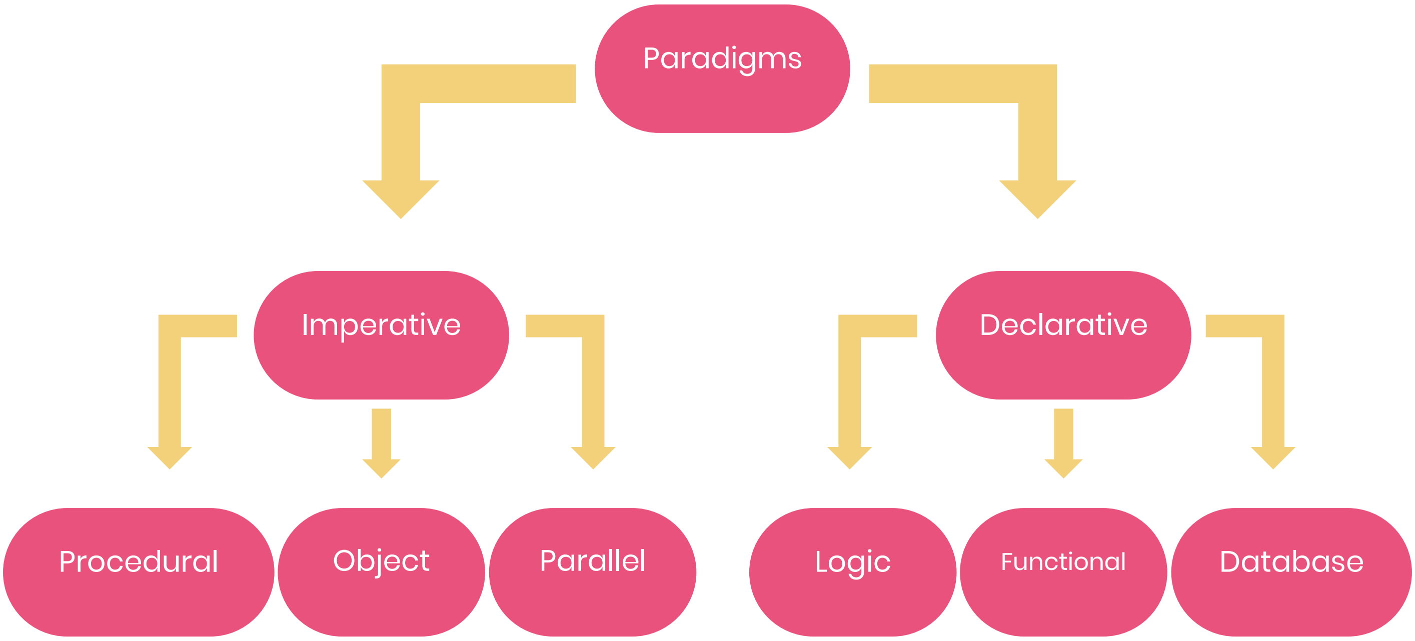
Types of programming languages

There are many types of programming languages and they are generally classified by their “level of difficulty”. This is, of course, subjective, as what’s difficult to one person may be very easy to another, just like learning to speak a language.

A programming language is a systematic method of describing a computational process. A computer can only really do arithmetic and logical operations, a number of control functions, input, and output. A programming language provides instructions for these basic computer operations.

As time progressed, programming languages have become more general and all-purpose, but each still has specific advantages, disadvantages, and best use cases.

Programming languages are classified based on their qualities and use cases



A programming language can either be imperative or declarative.

Imperative programming language

The imperative programming language paradigm has a close relationship with machine architecture. It works by changing the program state through assignment statements. It performs a task step by step, changing states. The main business of this approach is how the goal is achieved. Each step affects the global state of computation.

Imperative programming languages are simple to implement, but that comes at the cost of less efficiency and an inability to solve complex problems.

Also, parallel computing is not possible as each step affects the global state of the program and would result in spectacular chaos!

Imperative programming paradigm branches into:

- Procedural programming
- Object-oriented programming, and
- Parallel-processing programming.

Imperative programming paradigm branches

Procedural programming essentially executes a sequence of instructions that leads to a result. This approach typically includes multiple variables and heavy loops. Examples include C, C++, ColdFusion, Java and Pascal.

Object-oriented programming is based on the premise of viewing the world as a group of objects that have internal data and external access to parts of that data.

It is basically viewing a problem in parts and solving each part independently, then interconnecting the parts. Reusability is achieved by using inheritance and polymorphism (that's the ability to take on many forms, for those who don't know).

Parallel processing aims to speed up processing by distributing instructions among multiple processors. It's pretty much a 'divide and conquer'

Declarative programming

Declarative programming expresses the program in implicit terms. The focus here is what needs to be done, rather than how to do it.

There are no loops, assignments, etc. here, and there's not much difference between the specification of a program and how it is implemented. This allows instructions to be modelled at a very high level.

The declarative programming paradigm branches out into:

- Logic programming
- Functional programming, and
- Database programming, which we will look at more closely now.

In logic programming, programmers make declarative statements that allow the machine to reason about the consequences of those statements. In a way, this paradigm does not explicitly tell the computer what to do, but lets it consider what to do based on a number of restrictions. Think of what you would do if your friend says, “Bring me something nice.” You already know what your friend likes and doesn’t like, so you would make your decision based on those preferences.

Functional programming languages typically use stored data, preferring recursive functions over loops.

Their primary focus is pure mathematical functions and immutable data. In theory, you could change around the order of the code and still get the same result. Functional programming is most concerned with the flow of the program. Flow control is expressed by combining function calls, rather than assigning values to variables. This is where the power of this paradigm lies; passing functions to functions and returning functions from functions.

ReactJS is a good example of functional programming language. With functional programming, code is much shorter, less error prone and easier to correct.

The database programming theory is based on data and its movement. Program statements are defined by data rather than creating a series of steps.

A database is an organised collection of structured information, or data, typically stored electronically in a computer system. It is usually controlled by a database management system (DBMS) Most databases use Structured Query Language (SQL) for writing and querying data.

Databases can handle massive amounts of information, which can be accurately validated and updated using data manipulation languages |(DMLs). It is also easier to maintain data consistency.

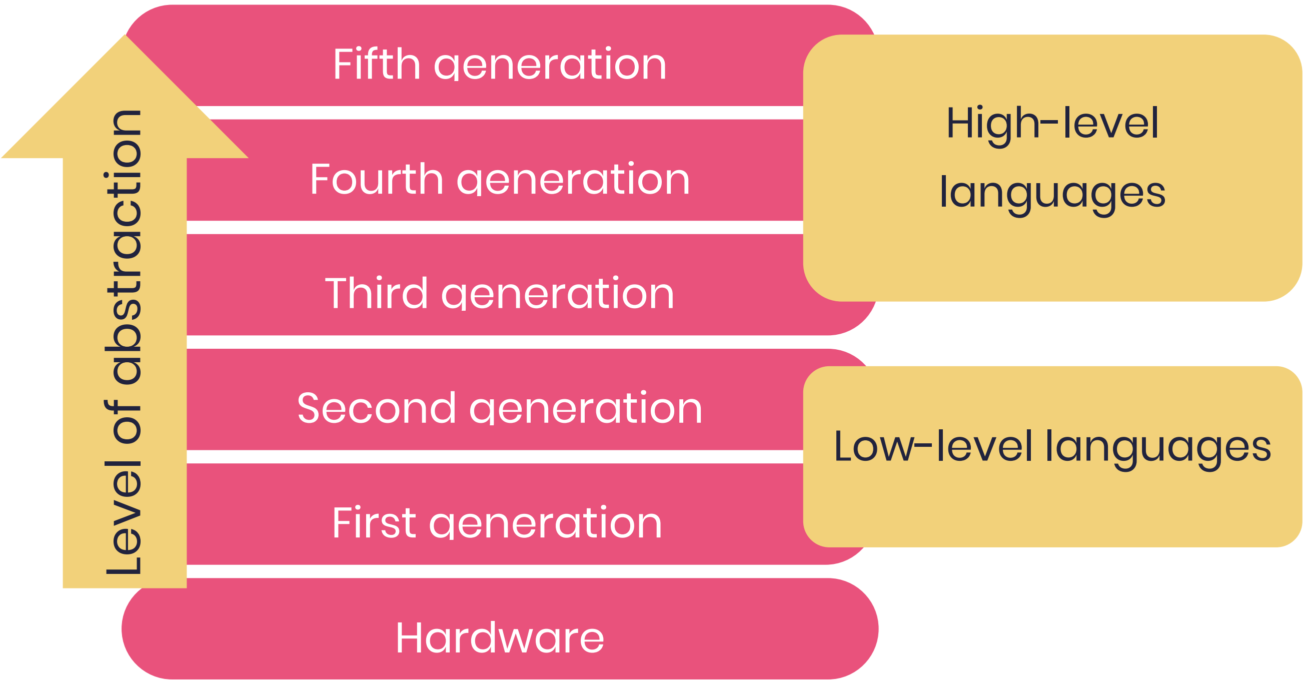
A final word on paradigm languages...

Very few languages implement a paradigm 100%. When they do, they are pure. It is incredibly rare to have a “pure OOP” language or a “pure functional” language.

A lot of languages will facilitate programming in one or more paradigms. If a language is purposely designed to allow programming in many paradigms it is called a multi-paradigm language. If a language only accidentally supports multiple paradigms, then there isn't any special name for that. It's formally referred to as a single paradigm language.

Generations of programming languages

The programming language timeline falls into several generations that distinctly define each group of programming languages. As you can see in this diagram there are 5 generations of programming languages and we will look at each one more closely now. As we move along the generation timeline, the programming languages become complex and more capable.



First generation languages

(1GLs) are the programming languages that were used to program the first generation of computers. Nothing more than a sequence of 1s and 0s, this language made it easy to understand the machine, but incredibly difficult to master. It required meticulous attention to detail and a superhero mind, which is impossible, of course, so errors were commonplace and took weeks to fix and get everything right. That said, here was no need for any translation, so machines read what the programmers wrote, which made the execution of these programs speedy and efficient.

All higher-level languages, fancy as they can be, are eventually humbled and translated back to machine code!

Machine code is machine specific. It cannot just be taken from one machine and run on a different one, because it interacts directly with the machine's transistors, so the code has to match the machine.

Second generation languages

(2GLs) use mnemonics in place of 1s and 0s to represent opcode and operands. Compared to machine code, it is a lot easier and less prone to errors. The machine, however, cannot directly read this code, known as assembly code, so it has to be dismantled into 1s and 0s by a program known as an assembler. Each processor family has its own assembler. Examples of assembly language programs include MIPS, X86, SPARC and RISC-V.

Third generation languages

(3GLs) have significant improvements compared to second generation languages. These came with the specific intention of making the programmer's life easier.

They moved away from the gobbledygook of 2GL English-like statements and English words for things like variables, data types and constants. This made it a whole lot easier to write code that you can follow, as though you were writing a story.

It also allowed for the programmer to define sections of code as subroutines. These languages generate a file called source code, which is converted by a program called a compiler into object code.

Object code is not quite machine code yet. It still goes through another process that links everything together to produce machine code. Don’t worry, we’ll look at this in more detail in a while. Third generation programming languages could run on many different types of computers, a property that is commonly referred to as portability.

Fourth generation languages

(4GLs) attempt to get closer to human language to reduce the overall time, cost and effort invested in software development.

The key to the efficiency of 4GLs is the use of icons, graphical interfaces, and symbolic representations, where appropriate.

4GLs are mostly associated with databases and data processing. (Examples of these include SQL (pronounced sequel), Postscript, Mathematica and AVS).

Fifth generation languages

(5GLs) are the highest level of languages so far.

They are based on the idea that a problem can be solved by building an application that uses constraints rather than approaching the problem algorithmically.

This means that, instead of determining how a solution is reached, the programming language is used to describe the properties of the logic of the solution. The computer is then left free to search for a solution that conforms to the specified constraints. This is hard for the computer, though, as ground-breaking as it sounds, and is not always suitable.

PROLOG (PROgramming LOGic) is an example of a 5GL. It uses a form of mathematical logic to solve queries on a database of facts and rules.

This class of programming language is at the centre of artificial intelligence and machine learning.

5GLs attempt to get the computer to solve a problem without the programmer. Essentially, the programmer just needs to worry about the conditions that must be met. This is music to the ears of computer scientists who specialise in making machines intelligent, because problems in that branch of computer science are not set in stone and can take a varied number of solutions.

We can summarise the generations of programming languages with the following table:

Generation	Programming language	Description
1st Generation	Machine code	Machine code is the 1s and 0s understood by a computer.
2nd Generation	Assembly language	Assembly language is made up of symbolic instructions and addresses that are basically placeholders for machine code. An assembler converts assembly code to machine code.

3rd Generation	Structured programming	Structured programming specifies a logical structure on the program being written to make it more efficient and easier to understand and modify. Structured programming typically uses the top-down design model, in which developers map out the entire program into separate subsections.
4th Generation	Non-procedural languages	Non-procedural languages focus on what users want to do rather than how they will be doing it. Better known as object-oriented programming. These languages make programming more concise than earlier languages.
5th Generation	Artificial intelligence languages	Artificial intelligence languages give the computer a certain degree of intelligence. These languages highly resemble natural speech.

Factors to consider when choosing a language

When developing software, you can't just pick your favourite language and use that for the task. Think of it this way, you wouldn't use excel to create a presentation, right? The wrong choice of programming language could work but may produce bloated, inefficient software that will eventually need to be rewritten in a more suitable language. And this costs time, effort and money, and sometimes even a company's reputation!

Programming paradigms help reduce the complexity of programs. A programmer needs to follow a language's paradigm approach when writing code.

Each paradigm has a use case – an area where it particularly shines, as discussed earlier.

Here's a set of some basic questions that can be asked when choosing a programming language:

- How readable is the language to humans?
- How easy is it to write the program in this particular language?
- How reliable is the language?
- What is the cost of developing in the particular program language?
- How complicated is the syntax going to be?
- Does the language have standards?

Of course, these are not set in stone. Lots of other questions can be asked, depending on the type of software and the target audience.

Language levels

You may have noticed on the timeline diagram that the generations of programming languages we have just discussed fall into two categories: low-level languages and high-level languages. Low-level languages are generally the earlier languages and high-level languages are the later ones.

Low-level languages

Low-level languages were the first to be developed. These are closely associated with the hardware they run on. This means that they are not portable, that is, they cannot be taken from one machine type to another machine type with the expectation of successfully running them.

Structure and function of low-level languages

There are two types of low-level programming languages: machine language and assembly language.

Machine language is directly run by the machine without the need for any translation. This results in very fast execution and efficient use of hardware. Because of their ability to interact directly with hardware and their efficient use of memory, they run faster.

An assembler translates assembly language to machine language, and the assembler itself is closely associated with the computer's hardware.

Low-level languages require extensive knowledge of the computer's hardware and its configuration. Low-level languages are by no means easy, check out this example:

```
8B542408 83FA0077 06B80000 0000C383

FA027706 B8010000 00C353BB 01000000

B9010000 008D0419 83FA0376 078BD98B

C84AEBF1 5BC3
```

That will get your head spinning! In case you're wondering, that is a function to calculate the Fibonacci sequence on an x86 machine! It would be borderline irritating to code a system for managing a hotel, for example. Now imagine making a mistake then having to look for it!

Use of low-level languages

The overall memory footprint of a low-level program is a lot smaller than that of a high-level language. This makes it suitable for systems with severe memory constrains.

Low-level languages are still used today to develop specialist code, such as that of device drivers and microcontrollers. There is very little abstraction from the hardware. This gives the programmer greater control of the hardware at the expense of ease of programming.

High-level languages

High-level languages are those from the 3rd generation onwards. They are more English-like and aim to make the programmer's life easier. They offer more hardware abstraction and so can be run on a variety of machines

Structure and function of high-level languages

High-level languages were created with the programmer in mind. The English-like statements enable the programmer to focus on the problem being solved, rather than what the machine is going to do with its transistors to solve the problem.

Knowledge of hardware configuration is not required, as a special program known as a compiler takes care of that. The human readable statements make it easier to find and "squash" bugs. In case you're wondering, bugs are errors that prevent a program from running as intended.

Writing English-like statements makes it a lot faster to learn coding and to write the actual code. Because of this, more people took up coding when high-level languages came into the picture. High-level languages also brought phenomenal advancements in things that computers can do, since people had more freedom to be creative with their code; programming languages that are easier to follow allowed programmers to lay out their thoughts more clearly.

High-level languages cannot be run by a computer as English-like statements. A translation software, either a compiler or interpreter, is required to convert the program into machine code. This process does not produce code that is as efficient as code written in machine code, so the program is slower than one written in machine code

Uses of high-level language

An important characteristic of high-level languages is their ability to be run on machines of different architecture.

Code written in high-level languages is compiled for a specific system. The same code written in C can be compiled by a windows compiler, then copied to a raspberry pi and compiled for ARM, and still run, producing intended results.

If you wanted to do this with low level code, you would need an emulator, that would emulate the original platform. An emulator is essentially a program that sits between the program you want to run, pretending to be the architecture intended for that program, while translating the code into instructions that are more suitable for the host machine. This would produce a really bloated system that is RAM hungry, generally inefficient, and pretty much not worth it. High-level languages have huge libraries and large sets of keywords, as well as many other features at the programmer's disposal.

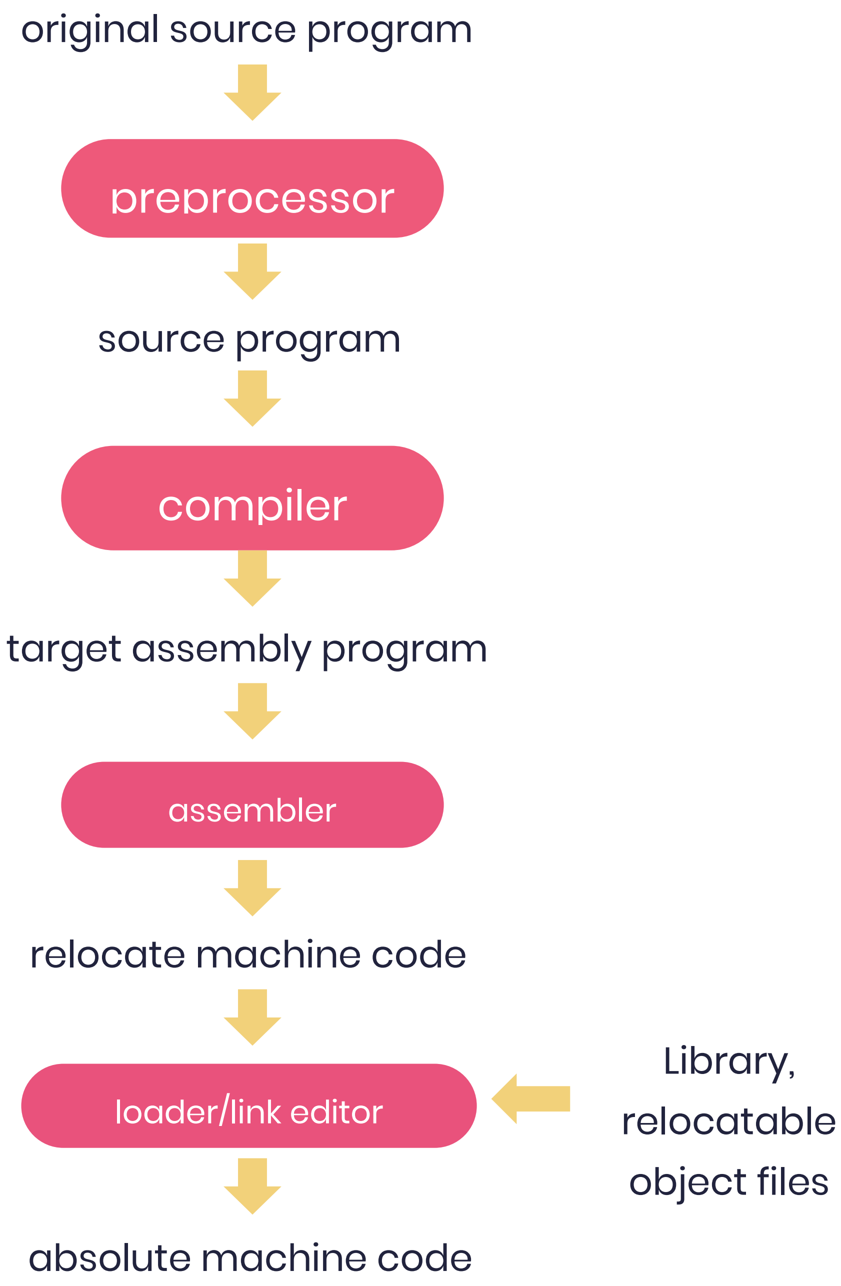
High-level languages	**Low-level languages**
Easy to learn and understand	Hard to learn and understand
Comparatively slower since they require translation	Faster execution because of direct relationship with hardware
Greater hardware abstraction	Little or no hardware abstraction
Not much hardware control	Great hardware control
Hardware knowledge is not required to code	Hardware knowledge is a requisite
Debugging is easy	Debugging is hard
Greater memory footprint	Smaller memory footprint
Portable	Not portable

The compilation processes

As we discussed previously, all languages from assembly language upwards need to be translated into machine language so that the machine can execute them.

This compilation process relies on 3 types of software: assemblers, compilers, and interpreters.

As mentioned earlier, compilers convert assembly code into machine code by converting mnemonics such as ADD and SUB back into their machine code equivalents. This is a pretty straightforward process, but for higher level languages, it's a bit more complicated.



As shown in this diagram, your program goes through the preprocessor, which will do a quick check on your code just before it goes to the compiler. It does a bit of formatting along with any instructions that you have given it. (In module 2, this is one of the things that we will be using a lot). After this, the code goes off to the compiler for conversion.

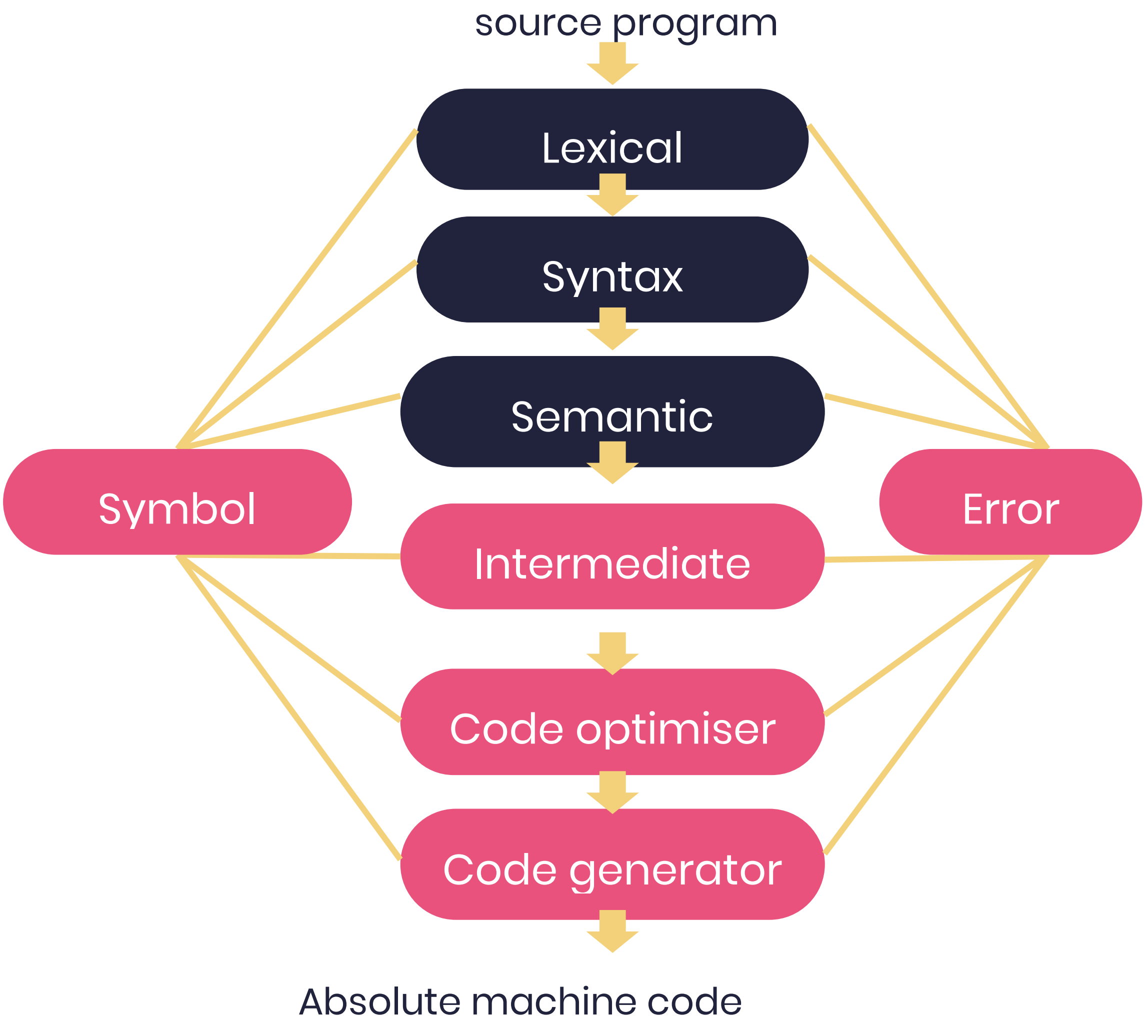
Compilers

Compilers convert high-level languages into machine code. Every integrated development environment (commonly known as IDE) that you will use has a compiler in the bundle. When you have finished writing your code, you then compile it for a specific target machine. The target machine is the hardware on which the program will run. The compiler will produce a file that contains machine code, called an executable, that the computer will execute.

Interpreters

Some high-level languages are not compiled, but rather have an interpreter. The end goal of the interpreter is the same as that of a compiler; it translates code into machine readable format. The difference is in the way that that is done; while compilers translate the whole program then execute it, interpreters translate the program line by line.

The compilation process



Compilation and interpretation follow fairly complex processes, which involve quite a number of steps. The compilation process uses the abstract machine models that we learned about in the previous lesson.

The compiler receives source code from the pre-processor.

It then starts with analysis, which comprises of three stages: lexical analysis, syntax analysis and semantic analysis.

Lexical analysis

The file is passed to the scanner for lexical analysis**.** This is the first phase.

The scanner reads your lines of code and breaks them down into a series of tokens, removing white space and other unnecessary things that were only just there to make code human readable, but not needed by computers. The tokens are a sequence of characters that represent a unit of information in the source program. If the scanner encounters an unrecognised symbol, it generates an error. Lexemes will be parts of the code taken as input.

Every time a token is created, the symbols are stored in the symbol table.

Syntax analysis

During syntax analysis, the scanner will check your code against the rules of the language.
Remember what we said about syntax in languages in lesson 4? This is what the syntax analyser checks in your code; to see if it follows the rules of the language.
If there are any violations, it will pass these to the error handler, and they will be presented to you. Compilation cannot proceed and the compiler will ask you to correct those errors.
If there aren't any errors, the code is passed over to the semantic analyser.

Semantic analyser

The semantic analyser looks at your code to see if it makes sense. For example, if you try to use a variable that you haven't declared. A variable is a portion of memory that you mention in your code so that it gets reserved for a particular set of data.

These steps will be linked to the symbol table. The symbol table is a data structure created by the compiler to store information about the various items in your code, such as variable names, function names, objects, classes, etc. That is how the compiler knows when you try to use a variable name that you never mentioned in your code.

Another important piece of software that is often referred to is the error handler.
All phases of the compiler are linked to the error handler, which reports all errors encountered during the compilation process.
The analysis stage produces intermediate code. This is not machine code yet, and still needs to go through a few more steps.
Here, we will be just about halfway through the process of generating machine code.
Our code as this point resembles assembly code. The code is then optimised in the code optimisation phase to further simplify it where possible, with the goal of generating smaller and faster machine code.

The code is then translated into target machine code. In this process, the compiler uses translation tables to translate each intermediate code instruction to the equivalent machine code. The output of this stage is called the target program.

After that pretty long detour explaining the compilation process, let’s come back to our path, this time with assembly code.
This now goes into an assembler, which will further translate the code to produce relocatable machine code. Assemblers usually do two passes on the code.

- The first pass consists of the assembler reading the source file code once. In this pass, the assembler takes note of the memory locations required and stores this information in a symbol table. This symbol table maintained by the assembler is different from the one used by the compiler.
- The second pass of the assembly process consists of the assembler reading the source file one more time and translating each operation code into the equivalent binary machine code, which the computer hardware can execute. The second pass of the assembly process outputs relocatable machine code.

Relocatable machine code just means code that can be executed in any relative memory location

The machine code now goes to the linker and loader.
The relocatable machine code is in several files and needs to be bundled together to produce one single executable file.
The linker will collate all the different bits of object code files; the linker searches through the object files and attempts to sort, export, and import all references to other modules. By resolving references, the linker includes the module, which contains the definition of the references within that object file.
After all references are resolved the linker terminates and hands over the next compilation task to the loader.

The loader is another software program that takes as input the single object module produced by the linker and loads it into the system. If the linker linked all files correctly and resolves all references, the loader loads the object file. If there are any references that the linker could not resolve, it will generate an error and the loader will not run.

After this long and complicated process is complete, we are left with an executable that consists of the actual 1s and 0s that are loaded into memory and executed by the host computer.
This code is referred to as the absolute machine code.
For every high-level language program that you are going to write, it will follow more or less the same process for it to be executable.
This file is no longer portable as it is compiled for that machine type. You cannot take a program that you have just compiled on a Windows machine, for example, which uses the x86 architecture and run it on a Windows machine that has ARM processor. The computer will not know what to do with the file!
This means that high-level language code is only portable while it is still in its uncompiled state.
This also means that if you write code, you need to compile it for each system that you intend to run it on. For example, while Microsoft Office works on windows, Mac, iOS and Android, the actual executable file sizes are wildly different, even if the features are exactly the same!
The machine’s instruction set, and many other factors affect the size of the resultant machine code, even if the source code is exactly the same.

Conclusion

We have covered the various types of programming languages.

We looked at how they evolved throughout the history of computers, and how they ended up falling into 5 categories. We also looked at programming paradigms and how we can use these to choose the correct programming language for the job.

Then we homed in on the two major levels of programming languages, as well as how languages are translated from human readable form to machine readable form.

References

F, H. (2019). Types of Programming Languages | Major Differences and Specialties | Temok Hosting Blog. [online] Temok.com. Available at: https://www.temok.com/blog/types-of-programming-languages/

3. Generations of Programming Languages. - Programming Languages (2020). 3. Generations of Programming Languages. - Programming Languages. [online] Google.com. Available at: https://sites.google.com/site/h810keith/generations-of-programming-languages

Angelfire.com. (2020). Though the idea of mechanizing mathematical thinking is not a new concept, the idea of enabling computers to be used by everyone. [online] Available at: http://www.angelfire.com/ultra/carolinechang/Compilers.htm

D.B. Naga Muruga (2019). Evolution of programming languages. [online] ResearchGate. Available at: https://www.researchgate.net/publication/337413763_Evolution_of_programming_languages

lamisa157724540 (2018). The Programming Language. [online] Lamisa Newaz. Available at: https://lamisa157724540.wordpress.com/2018/05/15/the-programming-language/

Lmu.edu. (2020). paradigms. [online] Available at: https://cs.lmu.edu/~ray/notes/paradigms/

Lmu.edu. (2020). paradigms. [online] Available at: https://cs.lmu.edu/~ray/notes/paradigms/

Mcmanis, C. (1997). Lexical analysis and Java: Part 1. [online] InfoWorld. Available at: https://www.infoworld.com/article/2076874/lexical-analysis-and-java--part-1.html

Oracle.com. (2020). What is a database? [online] Available at: https://www.oracle.com/database/what-is-database.html

Rungta, K. (2020). Lexical Analysis in Compiler Design with Example. [online] Guru99.com. Available at: https://www.guru99.com/compiler-design-lexical-analysis.html

Uct.ac.za. (2020). [online] Available at: https://www.cs.uct.ac.za/mit_notes/database/htmls/chp01.html.

www.javatpoint.com. (2011). Compilation Process in C - javatpoint. [online] Available at: https://www.javatpoint.com/compilation-process-in-c