

Types of translator

Learning objectives

evaluate the advantages and disadvantages of compilers and interpreters

Introduction

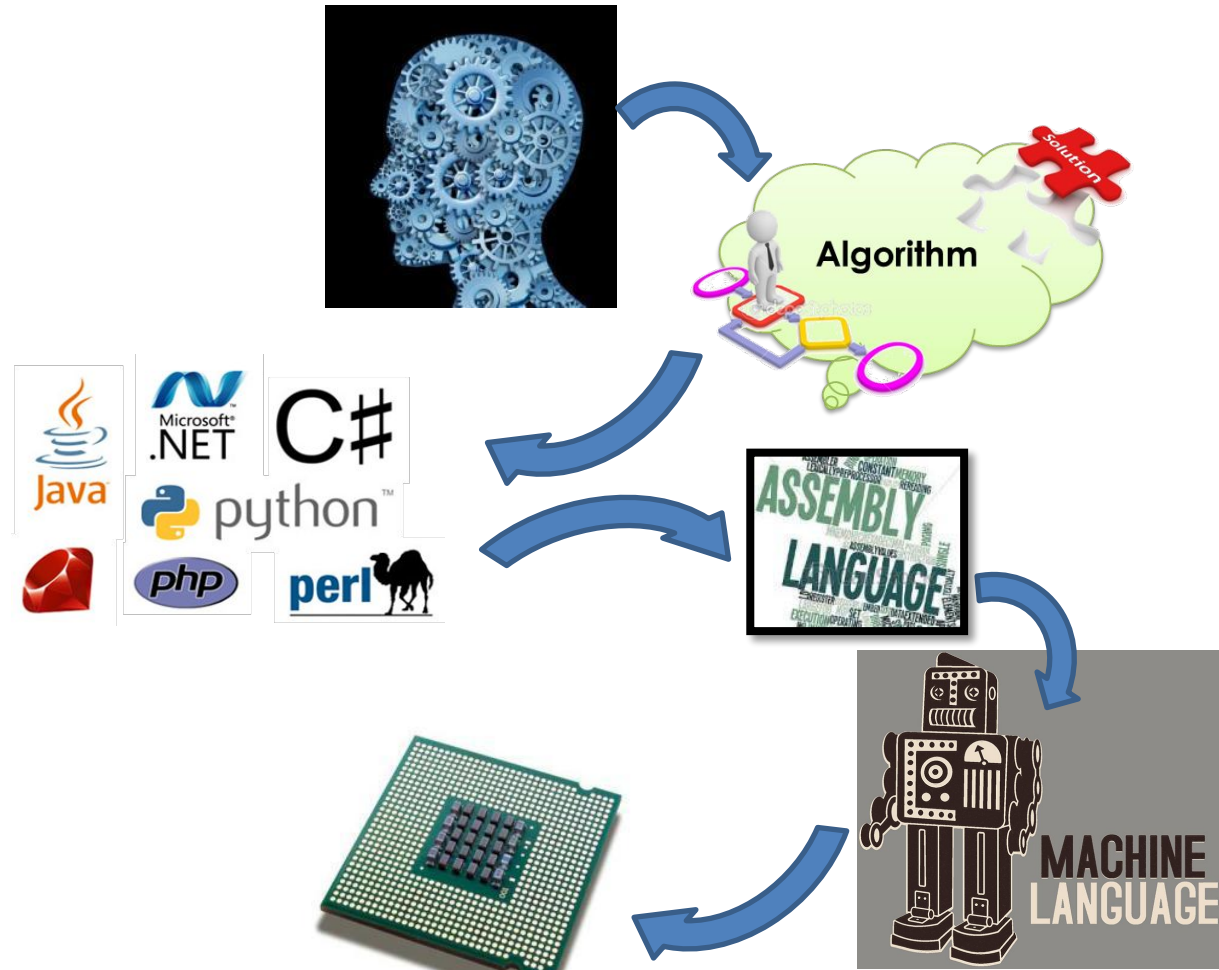
CPUs are very impressive **but** they are actually quite simple when it comes to processing.

They can only process 1's and 0's.

They therefore do **not** understand how to process programming code in the form in which we write it .

So **what has to happen** to get our code into a form that the CPU can work with...?

From Our Brain to the CPU...



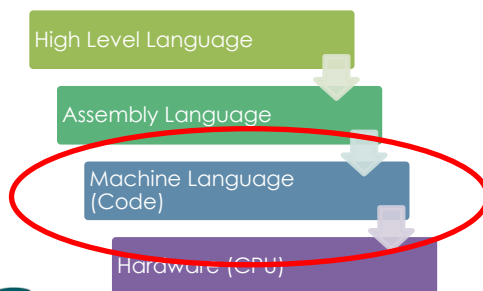
Difference between high level and low level code

Machine Code *(Very Low Level Language)*

The CPU can only understand one type of code: **Machine Code**

Made up of **Coded Instructions** and **Data**

e.g. 01001100 (binary)
Or
B8200 (hexadecimal)



Each piece of machine code is stored as a binary number and then decoded and executed by the CPU's logic circuits.

This means that writing programs in 'Machine Code' is difficult for a human to do.

Difference between high level and low level code...

Assembly Language (Low Level Language)

To overcome this issue, assembly language was created.

This language has a small set of commands which represent certain pieces of machine code

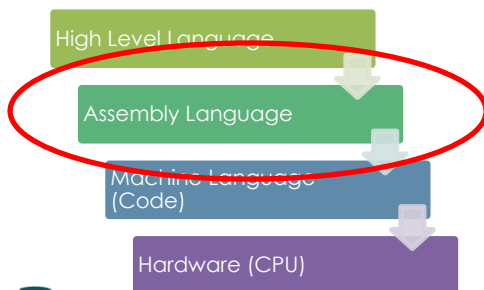
This helped programmers as they **didn't have to remember** sets of **binary code**, instead **they learnt commands**.

EG: Instead of memorising what 1011 or 1001 meant, they just had to remember commands like ADD and SUB

These commands are known as **Mnemonics** (simple memory aids).

Assembly language is still quite **difficult to learn and use** and this is why **High Level Languages were produced**.

(Due to less code, **assembly language** can be **processed by the CPU far more quickly** than **high level languages**)



Machine Code	Mnemonic Code
B80200	MOV AX, 0002
BB0400	MOV BX, 0004
01DB	ADD AX, BX
CD20	INT 20

Machine code (represented in Hex)
instructions are too difficult to remember...

Mnemonics make it easier for developers to
understand instructions & their meanings.

Difference between high level and low level code...

High Level Languages (*Python, Java, C++ etc*)

High level code (aka 'source code') is far more easy to write and therefore for humans to understand.

Its purpose is to be easier to write **AND STILL** be **easily translated** into machine code so that it can be processed by the CPU.

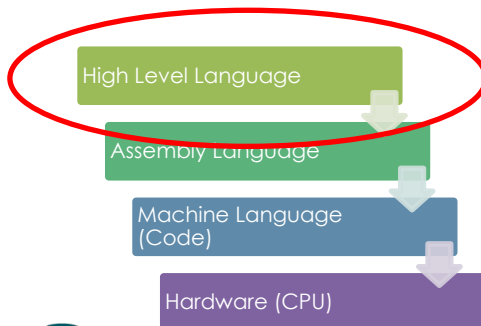
To help it be translated, it makes use of:

KEY WORDS

and

SYNTAX

(rules for the keywords and arguments that go with them)

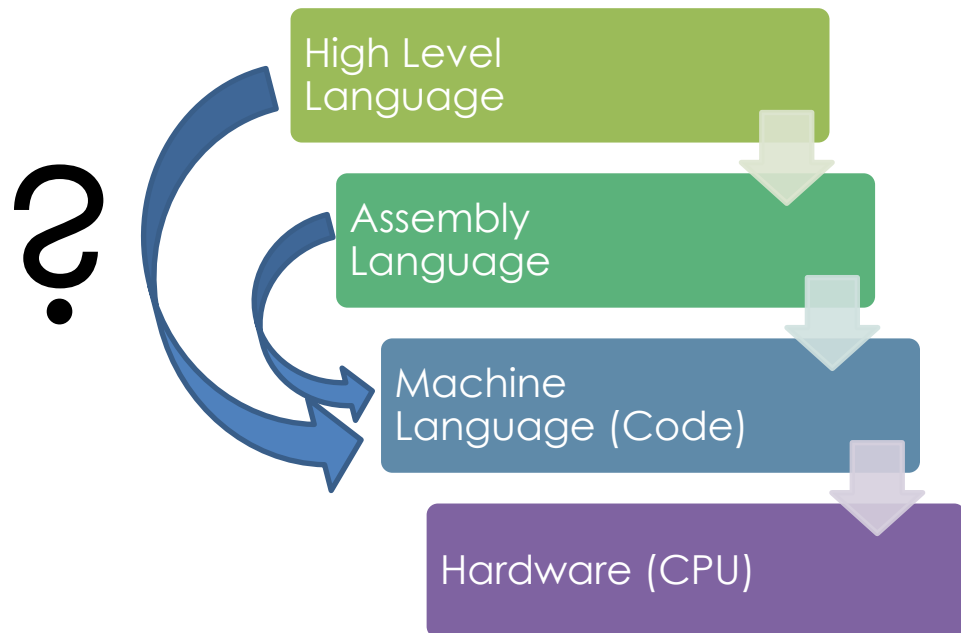


Using keywords as opposed to any old English means that translations can happen – *reserved words such as PRINT are known to translators where as 'put on screen' is not etc.*

Introducing Translators

So how does our '*HUMAN WRITTEN CODE*' get turned into Machine Code?

Translators!



Translators

Translators are programs that **convert high level language** commands:

print, IF, For etc.

...into a set of **machine code** commands:

1011, 11001, 11000011110 etc

...so that the CPU can process the data!

There are 2 ways in which translators work:

1. Take the **whole code** and convert it into machine code before running it (known as **compiling**).
2. Take the code **one instruction at a time**, translate and run the instruction, before translating the next instruction (known as **interpreting**).

Types of Translator

There are 3 types of translator:

Compiler

Converts the whole code into one file (often a .exe file).

The file can then be run on any computer without the translator needing to be present.

Can take a long time to compile source code as the translator will often have to convert the instructions into various sets of machine code as different CPUs will understand instructions with different machine code from one another.

Interpreter

Converts the source code into machine code 1 line at a time.

Program therefore runs very slowly.

Main reason why an interpreter is used is at the testing / development stage.

Programmers can quickly identify errors and fix them.

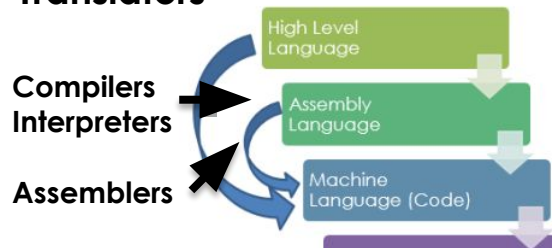
The translator must be present on the computer for the program is to be run

Assembler

This type of translator is used for Assembly Language (not High Level Languages).

It converts mnemonic assembly language instructions into machine code.

Translators



Assembler

An **assembler** translates assembly language into machine code. Assembly language consists of mnemonics for machine opcodes so assemblers perform a 1:1 translation from mnemonics to a direct instruction.

For example: LDA #4 converts to 0001001000100100

Conversely, one instruction in a high level language will translate to one or more instructions at machine level.

Advantages of using an Assembler:

- Very fast in translating assembly language to machine code as 1 to 1 relationship
- Assembly code is often very efficient (and therefore fast) because it is a low level language
- Assembly code is fairly easy to understand due to the use of English-like mnemonics

Disadvantages of using Assembler:

- Assembly language is written for a certain instruction set and/or processor
- Assembly tends to be optimised for the hardware it's designed for, meaning it is often incompatible with different hardware
- Lots of assembly code is needed to do relatively simple tasks, and complex programs require lots of programming time

Compiler

A **Compiler** is a computer program that **translates code** written in a high level language to a lower level language, object/machine code. The most common reason for translating source code is to create an executable program (converting from a high level language into machine language).

Advantages of using a compiler

- Source code is not included, therefore compiled code is more secure than interpreted code
- Tends to produce faster code than interpreting source code
- Produces an executable file, and therefore the program can be run without need of the source code

Disadvantages of using a compiler

- Object code needs to be produced before a final executable file, this can be a slow process
- The source code must be 100% correct for the executable file to be produced

Interpreter

An interpreter program executes other programs directly, running through program code and executing it line-by-line. As it analyses every line, an interpreter is slower than running compiled code but it can take less time to interpret program code than to compile and then run it — this is very useful when prototyping and testing code. Interpreters are written for multiple platforms, this means code written once can be run immediately on different systems without having to recompile for each. Examples of this include flash based web programs that will run on your PC, MAC, games console and Mobile phone.

Advantages of using an Interpreter

- Easier to debug(check errors) than a compiler
- Easier to create multi-platform code, as each different platform would have an interpreter to run the same code
- Useful for prototyping software and testing basic program logic

Disadvantages of using an Interpreter

- Source code is required for the program to be executed, and this source code can be read making it insecure
- Interpreters are generally slower than compiled programs due to the per-line translation method

Questions

- For a computer game running on a home console, would you use a compiler or an interpreter? Explain why.
- For a simple learning tool that will be released on the web for school children to use at home, what would be the best choice of translator? Explain why.

