IGCSE Computer Science — Unit 4.2

Types of programming language, translators and IDEs

Thomas Defranceschi

Made for IGCSE at Musashi International School Tokyo

Outline

What you need to be able to do

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

- Explain what **high-level** and **low-level** languages are
- Compare their **advantages and disadvantages**
- Explain **assembly language**, **mnemonics**, and the role of an **assembler**
- Describe how a **compiler** and an **interpreter** work, and how they report errors
- Explain the role of an **IDE** and common IDE functions

Outline

High-level vs low-level languages

High-level language (HLL)

- More **abstract** (further from hardware)
- Easier for humans to read and write
- Uses constructs like variables, loops, functions
- Needs a **translator** to run

Examples: Python, Java, C#

Low-level language (LLL)

- Closer to the CPU and hardware
- More direct control of memory/registers
- Often specific to a CPU architecture

Examples: machine code, assembly

HLL: advantages and disadvantages

Advantages

- **Easier to read and write** (closer to human language)
- **Easier to debug** (clearer structure, better tools)
- **Machine independent** (portable across different hardware)

HLL: advantages and disadvantages

Advantages

- **Easier to read and write** (closer to human language)
- **Easier to debug** (clearer structure, better tools)
- **Machine independent** (portable across different hardware)

Disadvantages

- Less **direct control of hardware**
- Can be less efficient in speed/memory than low-level code

Example (HLL): Python

```
a = int(input("Enter number 1: "))
b = int(input("Enter number 2: "))
print(a + b)
```

Example (HLL): Python

```
a = int(input("Enter number 1: "))
b = int(input("Enter number 2: "))
print(a + b)
```

- You don't manage the keyboard/screen directly
- `input()` / `print()` are **abstractions** that rely on the OS (system calls)

Example (HLL): Java

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(a + b);
    sc.close();
  }
}
```

Example (HLL): Java

```
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(a + b);
    sc.close();
  }
}
```

- Same task, different syntax: still **high-level**
- Input/output uses libraries + runtime + OS services (abstraction over system calls)

Which one is higher level: Python or Java?

Question: which one is higher level, Python or Java?

Which one is higher level: Python or Java?

Question: which one is higher level, Python or Java?

- **High-level** is a **category**, but it is also **relative**
- You can compare two languages and say one is **higher level** than the other

Which one is higher level: Python or Java?

Question: which one is higher level, Python or Java?

- **High-level** is a **category**, but it is also **relative**
- You can compare two languages and say one is **higher level** than the other

- In many contexts, **Python is considered higher level than Java**
- Reason: Python usually needs **less code/boilerplate** and hides more low-level details
- But **both are high-level** compared to assembly and machine code

LLL: advantages and disadvantages

Advantages

- **Direct manipulation of hardware**
- Can be **fast and memory-efficient**

LLL: advantages and disadvantages

Advantages

- **Direct manipulation of hardware**
- Can be **fast and memory-efficient**

Disadvantages

- **Hard to read and write**
- **Harder to debug**
- **Machine dependent** (not portable)

Example (LLL): Assembly (mnemonics)

```
; Read two numbers, add them, print result (illustrative)
READ  R1        ; input -> register R1
READ  R2        ; input -> register R2
ADD   R3, R1, R2
PRINT R3
```

Example (LLL): Assembly (mnemonics)

```
; Read two numbers, add them, print result (illustrative)
READ  R1        ; input -> register R1
READ  R2        ; input -> register R2
ADD   R3, R1, R2
PRINT R3
```

- Much closer to CPU operations (registers + instructions)
- Still readable (mnemonics), but not portable between CPU types

Example (LLL): Machine code (binary/hex)

```
; The *same idea* as the assembly slide,
;  but encoded for a CPU (illustrative)
10110000 00000001
10110000 00000010
00000001 00100001
11000000 00000011
```

Example (LLL): Machine code (binary/hex)

```
; The *same idea* as the assembly slide,
;  but encoded for a CPU (illustrative)
10110000 00000001
10110000 00000010
00000001 00100001
11000000 00000011
```

- This is what the CPU actually executes: patterns of bits
- Humans usually write assembly instead, then an assembler converts it

Which one is lower level: assembly or machine code?

Question: which one is lower level, assembly or machine code?

Which one is lower level: assembly or machine code?

Question: which one is lower level, assembly or machine code?

- **Low-level** is a **category**, but it is also **relative**
- **Machine code is lower level than assembly**
- Assembly is still low-level, but it is an **abstraction** (mnemonics) over binary

Which one is lower level: assembly or machine code?

Question: which one is lower level, assembly or machine code?

- **Low-level** is a **category**, but it is also **relative**
- **Machine code is lower level than assembly**
- Assembly is still low-level, but it is an **abstraction** (mnemonics) over binary

- You can also compare within languages: e.g. **C is often lower level than Python**
- The closer you are to registers/memory/instructions, the **lower level** it is

Abstraction example: variable names

Idea: high-level languages let you use **meaningful names**.

Abstraction example: variable names

Idea: high-level languages let you use **meaningful names**.

```
total = expenses + losses
print(total)
```

vs

```
; illustrative: you must remember what each location means
ADD   R3, R1, R2    ; R3 holds the result
PRINT R3
```

Abstraction example: variable names

Idea: high-level languages let you use **meaningful names**.

```
total = expenses + losses
print(total)
```

vs

```
; illustrative: you must remember what each location means
ADD   R3, R1, R2    ; R3 holds the result
PRINT R3
```

- In low-level code, you often track **numbers** (registers like `R3` or memory addresses)
- The programmer must **keep a mental map** of what each location represents (like a variable)
- Consequence: programs are usually **harder to read, write, and debug**

Outline

Assembly language and mnemonics

Assembly language: A low-level language that represents machine instructions using **mnemonics**.

Assembly language and mnemonics

Assembly language: A low-level language that represents machine instructions using **mnemonics**.

Mnemonic: A short, human-readable code for an instruction (e.g. ADD, SUB, MOV, JMP).

Assembly language and mnemonics

Assembly language: A low-level language that represents machine instructions using **mnemonics**.

Mnemonic: A short, human-readable code for an instruction (e.g. ADD, SUB, MOV, JMP).

Key idea: assembly is easier for humans than binary, but is still close to the CPU.

Why an assembler is needed

Assembly code (mnemonics)

$\Downarrow$ **assembler**

Machine code (binary)

Why an assembler is needed

Assembly code (mnemonics)

⇓ **assembler**

Machine code (binary)

- The CPU can only execute **machine code**
- The **assembler** translates mnemonics + operands into machine code

Assembly depends on the hardware

- Assembly is tied to a CPU's **instruction set**
- Different CPUs have different instructions, so they use different **mnemonics**
- Example: **x86** assembly and **ARM** assembly look different
- So **"assembly language"** is really a **family of languages**, not just one language like Python

x86	**ARM**	notes
`ADD`	`ADD`	completely equivalent
`MOV`	`MOV`	different operand
`LEA`	-	no direct equivalent

Outline

Compiler vs interpreter (big picture)

Compiler

- Translates the **whole program at once**
- Produces an **executable file**
- Then you run the executable

Interpreter

- Translates and executes **line-by-line**
- No separate executable in the same way
- Runs as it translates

Compiler vs interpreter: pipelines

Compiler pipeline

Source code

⇓ **compiler**

Executable file

⇓ **run**

Program output

Interpreter pipeline

Source code

⇓ **interpreter**

Executes line-by-line

⇓ **immediate result**

Program output

How errors are reported

Compiler

- Checks/translates the entire program
- Provides an **error report for the whole code** if errors are detected

How errors are reported

Compiler

- Checks/translates the entire program
- Provides an **error report for the whole code** if errors are detected

Interpreter

- Translates and runs one line at a time
- **Stops execution when an error is found**

Compiler

Advantages

- Compiles once, then can run many times (often faster overall)
- Produces an **executable** that can be distributed
- Can report **many errors** in one compilation attempt

Compiler

Advantages

- Compiles once, then can run many times (often faster overall)
- Produces an **executable** that can be distributed
- Can report **many errors** in one compilation attempt

Disadvantages

- Need to compile before running (slower edit-test cycle)
- Recompile after changes

Interpreter

Advantages

- Faster development: run immediately after editing
- Helps debugging: behaviour is easier to test step-by-step

Interpreter

Advantages

- Faster development: run immediately after editing
- Helps debugging: behaviour is easier to test step-by-step

Disadvantages

- Often slower execution (translates while running)
- Usually stops at the first error found

Interpreter

Advantages

- Faster development: run immediately after editing
- Helps debugging: behaviour is easier to test step-by-step

Disadvantages

- Often slower execution (translates while running)
- Usually stops at the first error found

Typical use: interpreter for development, compiler for final program.

A bit of context: choosing compiler vs interpreter

- In many cases, a programming language is **mostly compiled** or **mostly interpreted**
- So **"choosing between a compiler and an interpreter"** is usually not a choice you make *after* choosing the language
- The real choice is often: **which language/toolchain** are you using?
- Some environments may run scripts in a VM during development but ship a more **compiled/packaged** form
- Example (exception): in some workflows (like **GDScript** in Godot), scripts are executed via a VM during development, and exporting/shipping can bundle precompiled code/bytecode

Outline

What is an IDE?

IDE: Software that provides tools in one place to help you **write, run and debug** programs.

What is an IDE?

IDE: Software that provides tools in one place to help you **write, run and debug** programs.

Important: an IDE is not the same as a compiler/interpreter, but it often **includes** them.

Common IDE functions (you should know these)

- Code editor
- Run-time environment
- Translators (compiler/interpreter)
- Error diagnostics
- Auto-completion
- Auto-correction
- Prettyprint / formatting

Practice: match IDE functions and definitions (1/2)

Auto-completion	Runs your code
Run-time environment	Converts source code to machine code
Translators	Suggests words/identifiers while you type
Error diagnostics	Shows errors/warnings and where they are

Practice: match IDE functions and definitions (2/2)

Code editor	Corrects small typing mistakes
Auto-correction	Makes code layout consistent
Prettyprint	Lets you write/edit program text

Outline

Common exam phrasing to include

- Low-level languages are **hard to read** but allow **direct hardware control**
- High-level languages are **easier to write/debug** and are **machine independent**
- Assembly uses **mnemonics**; an **assembler** translates it to machine code
- A **compiler** translates the whole code and can give an error report for the whole program
- An **interpreter** translates/executes line-by-line and stops at the first error

Outline

Practice 1: quick definitions

Try first (no notes): give a short definition for each.

- High-level language (HLL)
- Low-level language (LLL)
- Translator
- IDE

Practice 1: quick definitions

Try first (no notes): give a short definition for each.

- High-level language (HLL)
- Low-level language (LLL)
- Translator
- IDE

Answers (short)

- HLL: more abstract; easier for humans; needs translation to run
- LLL: closer to hardware (registers/instructions); often machine-dependent
- Translator: software that converts/executes code so it can run on a CPU
- IDE: software that provides tools to write/run/debug programs in one place

Practice 2: compare HLL and LLL

Question: State **two advantages** and **one disadvantage** of:

- High-level languages
- Low-level languages

Practice 2: compare HLL and LLL

Question: State **two advantages** and **one disadvantage** of:

- High-level languages
- Low-level languages

Possible answers

- HLL advantages: easier to read/write; easier to debug; portable (machine independent)
- HLL disadvantage: less direct hardware control; can be less efficient
- LLL advantages: direct hardware control; fast and memory-efficient
- LLL disadvantage: hard to read/write/debug; machine dependent

Practice 3: abstraction (variable names)

Question: Explain what **abstraction** means using the idea of **variable names vs registers/memory addresses**.

Practice 3: abstraction (variable names)

Question: Explain what **abstraction** means using the idea of **variable names vs registers/memory addresses**.

Answer (short)

- HLLs let you name values (e.g. `total`), hiding hardware locations
- In low-level code you often track numbers (registers like `R1` or memory addresses)
- This makes programs harder to read and increases the mental load on the programmer

Practice 4: assembly and the assembler

Question: Explain:

- What a **mnemonic** is
- Why an **assembler** is needed
- Why assembly is called a **family** of languages

Practice 4: assembly and the assembler

Question: Explain:

- What a **mnemonic** is
- Why an **assembler** is needed
- Why assembly is called a **family** of languages

Answers (short)

- Mnemonic: short readable instruction code (e.g. ADD, MOV)
- Assembler: converts assembly (mnemonics + operands) into machine code that the CPU executes
- Assembly depends on the CPU's instruction set (ISA): x86 and ARM use different mnemonics/instructions

Practice 5: compiler vs interpreter (process + errors)

Question: Compare a compiler and an interpreter in terms of:

- How translation happens
- What happens when an error is found

Practice 5: compiler vs interpreter (process + errors)

Question: Compare a compiler and an interpreter in terms of:

- How translation happens
- What happens when an error is found

Answers (short)

- Compiler: translates whole program; produces executable; can report many errors at once
- Interpreter: translates and executes line-by-line; stops when it finds an error

Practice 6: exam-style short questions

Try to answer, then reveal.

- Give **three** IDE functions and what they do.

Practice 6: exam-style short questions

Try to answer, then reveal.

- Give **three** IDE functions and what they do.

Answers (examples)

- Code editor: lets you write/edit program text
- Run-time environment: runs your code
- Translators: converts source code to machine code
- Error diagnostics: shows errors/warnings and where they are
- Auto-completion: suggests words/identifiers while you type
- Auto-correction: corrects small typing mistakes
- Prettyprint: makes code layout consistent