

Cambridge IGCSE · Computer Science 0478 · Notes

Programming languages, translators and IDEs

Read online at pastopic.com/cambridge/computer-science-0478/notes/programming-languages-and-translators

Last checked 30 September 2026

Read first: [Computer architecture](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Say what high-level and low-level languages are** and weigh them up against each other: which is quicker to read and write, which is easier to find mistakes in, which can be moved between different processors, and which gives the programmer direct control of the hardware.
2. **Recognise assembly language** as a kind of low-level language written with short codes called mnemonics, and know that a program called an **assembler** turns it into machine code.
3. **Describe what a compiler does and what an interpreter does**: how each one gets a high-level program into a form the CPU can run, and what each one does when the code contains mistakes.
4. **Explain what is good and bad about each of the two**, and why programmers tend to use an interpreter while a program is still being built and a compiler once it is finished.
5. **Explain how an integrated development environment (IDE) helps a programmer**, and what its usual tools do: the code editor, the run-time environment, built-in translators, error diagnostics, auto-completion, auto-correction and prettyprint.

Papers from 2021 and 2022 sometimes tack on parts about freeware, free software, shareware, copyright and plagiarism. Those belong to an older syllabus and are not part of this one.

Understand it

Why programs need translating

The only thing a CPU can carry out is **machine code**: instructions made of binary digits, such as 1011001100001010. Each processor family has its own instruction set, so machine code for one kind of chip means nothing to another. Writing long programs in binary is slow and error-prone, so programmers usually write in something friendlier and let a **translator** (itself just another program) produce the machine code.

Languages fall into two families.

High-level languages

A **high-level language** reads a bit like English, with words such as `IF`, `WHILE` and `OUTPUT` and names the programmer chooses. Python and Java are examples, but in an exam answer write "high-level language", never a brand name. A single statement can stand for several machine steps: `area = width * height` means "fetch width, fetch height, multiply them, put the answer in area", and the programmer never has to think about which memory addresses or registers are involved.

Good points:

- it is **machine independent (portable)**: the same source code can be translated for many different processors;
- it hides the details of memory and registers, so the programmer's attention stays on **what the program should do**;
- the code is **easier to read, write and understand**, because it uses English-like statements and meaningful names;
- mistakes are **easier to find and fix** (easier to debug), and fewer are made in the first place, so programs get written faster.

Bad points:

- it has to be **translated into machine code** before it can run, which takes time;
- it **cannot directly manipulate the hardware** (a particular register or memory address) and cannot use special instructions that only one processor has;
- the machine code a translator produces is usually **bigger and slower** than code written by hand at a low level, so it uses more memory.

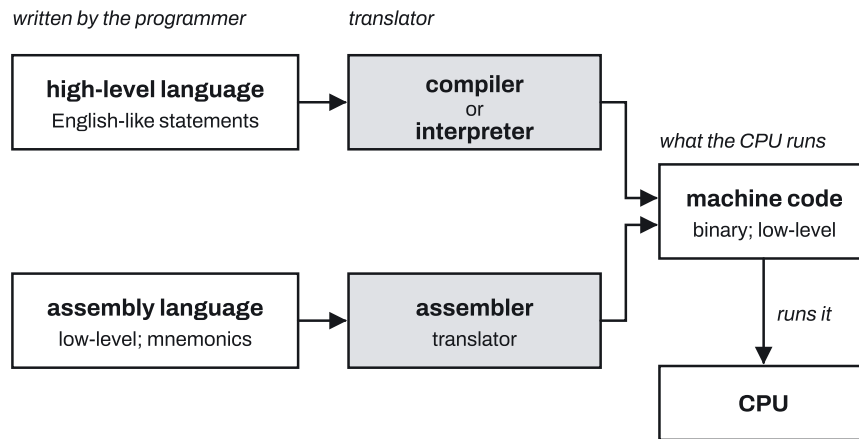
Low-level languages

A **low-level language** is close to what the processor itself understands. There are two: **machine code** (the binary itself) and **assembly language**. Low-level code is tied to one type of processor.

Assembly language replaces each binary instruction with a **mnemonic**, a short memory-jogging code. Here is `area = width * height` again, now in an invented assembly language:

Line	Mnemonic	What it does
1	LDA width	Copy width into the accumulator
2	MUL height	Multiply what is in the accumulator by height
3	STO area	Copy the accumulator's value into area

Each mnemonic normally matches **one** machine code instruction, so every line does just one small job. Before the CPU can use it, an **assembler** converts the mnemonics into machine code. Machine code itself needs no translator at all.



Good points:

- the finished program **runs quickly**, and it is **small** and **memory efficient**;
- the programmer gets **direct control of the hardware**: particular registers, memory addresses and devices;
- it can use **instructions and hardware features special to that processor**;
- machine code needs no translator.

Bad points:

- it is **hard to read and write**, and hard to learn;
- mistakes are **hard to track down** (difficult to debug);
- it is **not portable**: moved to a different type of processor, it has to be rewritten;
- one line does very little, so programs are long and take a long time to write.

Picking one. Low-level suits small programs that talk straight to hardware, must fit in very little memory or must react instantly, and will stay on one chip: the software inside a sensor, a household appliance or a device driver (an **embedded system**). High-level suits nearly everything else, especially big programs that will be changed often or must run on many kinds of device, such as apps and games.

Compilers and interpreters

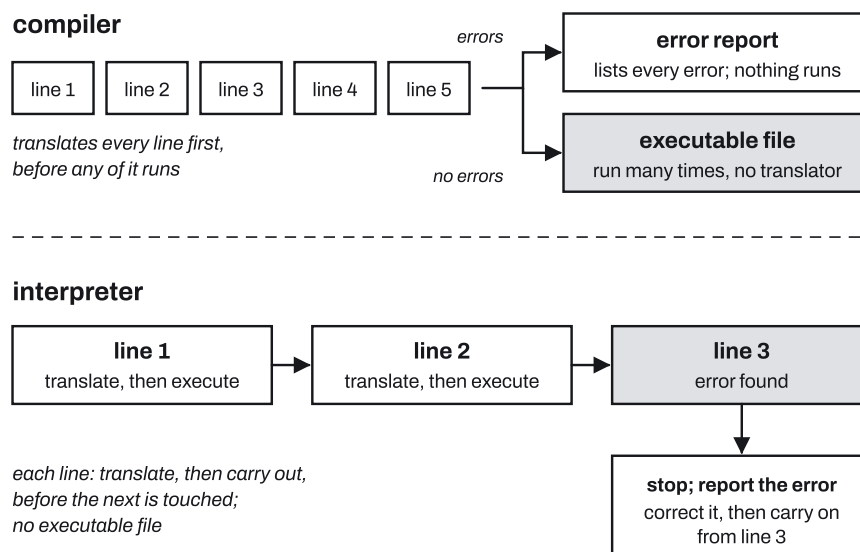
High-level code can be translated by either of two translators. What they share: both turn high-level code into machine code (sometimes called object code), and both detect mistakes and tell the programmer about them. What differs is the order in which they do things.

A compiler works on the whole program in one go:

1. every line of the source code is translated before **any** of it is run;
2. if there are mistakes, nothing runs; instead the programmer gets **one error report covering every error** found in the whole program;
3. once there are no errors, the result is an **executable file** of machine code. That file can be run as often as you like, on its own, with no compiler and no source code needed.

An interpreter works line by line:

1. each line is translated and **carried out immediately**; only after that is the following line looked at;
2. when it reaches a line with a mistake, it **stops there** and reports that one error;
3. the programmer fixes the line and the program **carries on from where it stopped** (errors are fixed in real time);
4. it makes **no executable file**, so every run means interpreting the code again, and anyone who runs it needs the interpreter.



Precision matters here. With a compiler, the translating is **completely finished** before a single line runs. With an interpreter, translating and running **alternate**, one line each time.

Which translator, and when

	Interpreter	Compiler
Strengths	You can try out part of a program before the rest exists; a problem is pinpointed, because it halts at the faulty line; after the fix it resumes from there, with nothing retranslated	All the errors arrive together in one report; the result runs faster; users need only the executable file, not a translator, and nothing is retranslated on each run; the source code can stay private, since only the executable is handed out
Weaknesses	No executable file, so the code is translated again on every run and users need the interpreter; the source code has to go out with the program; it runs more slowly	Harder to debug: nothing at all runs until every error is fixed; after each fix the whole program must be compiled again
Usually used	During development: writing, testing and fixing the program	For the finished program: handing it out, and running the same tests again and again

In short: **interpret while you build it, compile when it is done.**

Integrated development environments (IDEs)

An **integrated development environment (IDE)** is one program that bundles together all the tools for producing software, so the programmer can type code, translate it, run it and hunt down mistakes without switching between separate programs. That saves time and cuts down on errors. The tools it usually offers:

Tool	What it does for the programmer
Code editor	The window where code is typed in, changed and saved
Run-time environment	Launches the program inside the IDE and shows what it prints, so you can test it without leaving
Translator	A compiler and/or interpreter built in, to turn the code into machine code
Error diagnostics	Points out where mistakes are, with a message saying what is wrong, so they are quicker to fix
Auto-completion	Offers a list of ways to finish the keyword or name you have started typing; you pick one
Auto-correction	Quietly fixes small typing slips in keywords, so <code>PRNIT</code> becomes <code>PRINT</code>
Prettyprint	Colours keywords, names and comments differently (and lines up indentation) so the code is easier to read

You may also meet tools that fold a block of code away and open it again, and tools that produce documentation automatically. When an answer asks for a tool's role, say what it **does for the programmer**; repeating its name ("auto-completion completes the code") earns nothing.

Key facts and formulas

There are no formulas in this topic, and nothing is given in the exam. These are the facts to learn.

Fact	Status
High-level: English-like statements; easy to read, write and debug; machine independent (portable); must be translated; can't directly manipulate hardware	Learn it
Low-level: machine code or assembly language; directly manipulates hardware; memory efficient and fast; hard to read, write and debug; not portable	Learn it
Assembly language uses mnemonics and is translated by an assembler	Learn it
The three translators: compiler and interpreter (for high-level code), assembler (for assembly language)	Learn it
Compiler: translates everything before anything runs; one report of all errors; produces an executable file	Learn it

Fact	Status
Interpreter: translates and runs one line at a time; halts at an error; carries on after the fix; no executable file	Learn it
Interpreter during development; compiler for the finished program	Learn it
IDE tools: code editor, run-time environment, translators, error diagnostics, auto-completion, auto-correction, prettyprint	Learn it

How to do it

In the Paper 1 questions we have tagged for this topic (20 questions, 2021–2025), the types came up this often. 13 of the 20 questions mix more than one type, so the counts add up to more than 20.

Question type	Questions
High-level and low-level languages: meaning, advantages and choosing	11
How a compiler and an interpreter work	9
Assembly language and the assembler	6
When to use an interpreter or a compiler	5
IDE functions	5

1. High-level and low-level languages: meaning, advantages and choosing (11 questions)

- Meaning of a high-level language** (1–3 marks). Say what it **is**, one fact per mark: one statement can do several jobs; the CPU can't run it until a translator has turned it into machine code; it is portable; it reads like English.
- Meaning of a low-level language** (2 marks): its instructions match, or nearly match, the processor's own instruction set; it may be written with mnemonics; assembly language and machine code are the two kinds.
- Advantages and disadvantages** (1 mark each): pick from the lists in "Understand it". Check which language the question is asking about, and don't hand back a point the question already gave you.

Variations you will meet:

- One difference between high-level and low-level** (2 marks): make it a matched pair, one side each, linked with "whereas". For example: "high-level code can be translated for any processor, **whereas** low-level code is written for one type of processor." Other pairs: English-like statements / mnemonics or binary; easy / hard to read, write and debug; can't / can directly manipulate the hardware; one line does several actions / one line does one action.

- **A tick box asking for a high-level benefit.** The right box is about portability (machine independence) or easy debugging. Direct hardware control, better memory use, faster running and mnemonics are all **low-level** features. Examiners report that candidates often think "high" means "harder"; it actually means further from the hardware and nearer to people.
- **Why a device's software might be low-level** (a sensor, an appliance, anything embedded): start from what the device needs. Tiny memory: low-level programs are compact. Instant reaction: they run fast. Its own chips to drive: registers and ports can be controlled directly, including features only that processor has. Never moving to another chip: losing portability costs nothing. (Machine code also skips translation altogether.)
- **Choose a language for a setting and justify it** (about 4 marks). The choice itself scores nothing, so either can do well if the reasons fit. Link each reason to its effect. High-level: portable **so** it runs on any device; easier to debug **so** faults are sorted out faster; quicker to write **so** the program is ready sooner. Low-level: access to special hardware; a faster response because the hardware is controlled directly; a smaller memory footprint.
- **"Which type of language uses English-like statements?"** High-level language.
- **A tick table of statements against high-level language, assembly language and machine code** (some rows take two ticks):

Statement	Ticks
needs a translator	high-level and assembly
is low-level	assembly and machine code
uses mnemonics	assembly
uses English-like statements	high-level
can directly manipulate the hardware	assembly and machine code
is portable	high-level

2. How a compiler and an interpreter work (9 questions)

1. **Name the translator from a description** (1 mark each). Anything about an **executable file**, a report of **all** errors, or translating **everything before running** → compiler. Anything about **one line at a time**, **halting** at an error, or **carrying on** after a fix → interpreter.
2. **What is high-level code translated into?** Machine code (low-level language or object code also fit).
3. **How a compiler translates** (about 3 marks): an executable file comes out at the end; all the source code is translated before any of it runs; the output is machine code.
4. **How a compiler reports errors** (about 2 marks): nothing runs until they are fixed; every error in the program goes into one report; the report appears after the compiler has tried to translate the whole thing.

5. **How both work** (2 marks each): for the compiler, pick from executable file, all errors in one report, everything translated before anything runs. For the interpreter, pick from each line carried out as soon as it is translated, halt at an error, carry on from the same place after the fix.

Variations you will meet:

- **A similarity:** both detect errors; both report errors; both turn high-level code into machine code. Translators **find and report** mistakes; they don't fix (debug) them.
- **Two differences** (4 marks): each difference needs **both** translators, a mark for each side. Good pairs: runs up to the error **whereas** runs nothing if there are errors; needed on every run **whereas** not needed once compiled; halts at the first error **whereas** one report of all errors at the end; only a compiler leaves behind an executable file; line by line **whereas** everything at once. A list of facts about just one translator is not a difference, however many facts it has.
- **"Compare the features":** include what the two have in common, not only how they differ. Stick to what each translator actually **does** (it writes an executable file), rather than the knock-on benefit for users.
- **Gap fills from a word list:** the words that keep appearing are high-level language, whole code (or all at once), executing, all (errors), line by line, error, executable file and is not required.
- **"Identify one other type of translator"** once compiler and interpreter are used up: assembler.

3. Assembly language and the assembler (6 questions)

1. **What assembly language is** (2 marks): one of the low-level languages; its instructions are short mnemonics such as LDA; it gives orders straight to the processor and its registers, with nothing in between.
2. **Its translator:** assembler, and nothing else.
3. **High-level, low-level or machine code?** Low-level. It is **not** machine code: an assembler has to convert it first.

Variations you will meet:

- **"Machine code is one low-level language; name another":** assembly language (translated by an assembler). "Binary" is wrong, because binary is machine code.
- **Why use assembly language** for a device: the same device-needs reasoning as the low-level bullet in type 1. In a tick box, the advantage is "memory efficient"; "close to human language", "easy to debug" and "machine independent" are all high-level points.
- **Downsides of assembly language:** a program written for one chip has to be redone for another; mnemonics take effort to learn and to read; bugs are hard to track down.

4. When to use an interpreter or a compiler (5 questions)

1. **When** (1 mark): interpreter while the program is being written, tested and fixed; compiler once it is finished, to hand it out or to rerun tests.
2. **Why** (2 marks), each point with its effect:
 - interpreter: see the effect-first reasons just below;

- compiler: the source code stays private, **because** only the executable is handed out; users' machines need no translator, **because** the executable already is machine code; test after test runs from the same executable, **so** there is no retranslating.

Variations you will meet:

- **"Why is an interpreter better while developing?"** (about 3 marks). Lead with what the programmer gains, then the reason. *Finding a bug is quick*, because execution halts at the faulty line and that line is reported. *Fixing it is quick*, because after the correction the program resumes from that line rather than the whole program being translated again. *Unfinished work can be tried*, because each line is handled as it is reached, so a half-written program still runs as far as it goes.
- **"Advantages of an interpreter over a compiler during development"** (about 4 marks): explaining the mechanics of both translators scores little here; give gains for the programmer. Be precise about errors: the interpreter **halts** at the first one it meets and goes no further until it is fixed. "It shows the errors one by one" sounds like a list, so avoid it.
- **A different translator for each task, with benefits:** releasing the finished program without its source code → compiler (the source stays private, and buyers run the executable with no translator); building and testing → interpreter (each error shows up the moment its line is reached, which makes debugging easier).
- **"When is each most appropriate?"** Give the stage as well as the reason. "It is easier to debug" is worth less without "so it is used while developing".

5. IDE functions (5 questions)

1. **Purpose of an IDE** (about 4 marks), one idea per mark: name an example tool (error diagnostics, say); it tracks down mistakes; it turns code into machine code with a built-in translator; it runs programs so they can be tested; and it is a single piece of **software** in which the code is also typed and edited.
2. **Name some IDE tools** (1 mark each): code editor, run-time environment, error diagnostics, auto-completion, auto-correction, prettyprint, and translators unless the question has already used them.
3. **A tool with its role, or ways an IDE helps:** a mark for the tool and a mark for what it does, taken from the table in "Understand it". The role has to add information: "auto-correction fixes a mistyped keyword such as PRNIT to PRINT".

Variations you will meet:

- Don't give **operating system** jobs (managing memory, handling interrupts); examiners report this. Every IDE tool is about writing, running or fixing code.
- If the question already says the IDE provides a translator, give **other** tools.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

These two fragments of code do the same job.

Fragment P:

```
pay = hours * rate + bonus
```

Fragment Q:

```
LDA hours  
MUL rate  
ADD bonus  
STO pay
```

- (a) Name the type of language each fragment is written in. **[2]**
- (b) Using the fragments, give **two** reasons why a programmer could write and check fragment P more quickly than fragment Q. **[2]**
- (c) Neither fragment can be run by the CPU as it is. State the form each must be changed into, and name a translator that could change each fragment. **[3]**
- (d) Both fragments are copied to a computer with a different type of processor. Explain which fragment is more likely to work there without being rewritten. **[2]**

(a) P: a high-level language **B1**. Q: assembly language (a low-level language) **B1**.

(b) Any two:

- P is one statement doing the work of four, because one high-level statement can carry out several actions **B1**;
- P reads like English, with a familiar formula, so mistakes in it are easier to spot and fix **B1**;
- Q forces the programmer to think about the accumulator and the order of machine steps, which P hides **B1**.

(c) Both must become machine code **B1**. P: a compiler or an interpreter **B1**. Q: an assembler **B1**.

(d) Fragment P, because a high-level language is machine independent: the same line can be translated for the new processor **B1**. Q's mnemonics belong to one processor's instruction set, so assembly language is not portable and would have to be rewritten **B1**.

Example 2

A sports equipment company is writing two programs.

- **Program S** runs on the chip inside a cycling helmet's crash sensor. The chip has only 16 KB of memory, and the program must read the sensor's hardware registers and react within a few milliseconds.
- **Program T** shows riders their ride history. It must run on many kinds of laptop and phone, and new features will be added every few months.

(a) Recommend a type of programming language for each program. Justify each recommendation using that program's requirements. **[6]**

(b) Two years later the helmet uses a new chip with a different processor. Explain one effect of your recommendation for Program S. **[2]**

(a) No marks for the choices themselves; up to 3 marks for each program's reasons.

Program S: a low-level language (assembly language).

- It is memory efficient and makes small programs, so the code fits in 16 KB **B1**;
- it can directly manipulate the hardware, so it can read the sensor's registers itself **B1**;
- it runs quickly, so the helmet can react within milliseconds **B1**.

Program T: a high-level language.

- It is machine independent, so one program can be translated for each kind of laptop and phone **B1**;
- it is easier to read, write and debug, so new features can be added quickly and with fewer mistakes **B1**;
- nobody has to deal with each device's memory and chips, so the effort goes into the ride-history features **B1**.

(b) Assembly language is written for one type of processor, so it is not portable **B1**; Program S would have to be rewritten (and assembled again) for the new chip **B1**.

Example 3

A program written in a high-level language has 40 lines. Line 12 and line 31 each contain an error; every other line is correct.

(a) The programmer gives the program to a compiler. Describe what the compiler does, and what the programmer sees. **[3]**

(b) The programmer gives the original program to an interpreter instead. Describe what happens. Your answer should state which lines have already been executed when the first error message appears. **[3]**

(a)

- It translates all 40 lines before any of them is executed **B1**;
- it produces one error report that lists both errors, in line 12 and line 31 **B1**;
- no executable file is produced, and none of the program runs until both are corrected **B1**.

(b)

- Each line is translated and carried out immediately, one line at a time **B1**;
- lines 1 to 11 (11 lines) have been executed when it stops at line 12 and reports that error only **B1**;
- once line 12 is corrected it carries on from line 12, runs until it stops at line 31, and so on; no executable file is produced **B1**.

Example 4

A small team looks after a school's timetable program, which is written in a high-level language.

(a) For each job, choose the more suitable translator (compiler or interpreter) and give a reason linked to the job. **[6]**

- (i) Trying out a changed procedure while other parts of the program are still half-written.
- (ii) Installing the corrected program on 30 office computers that have no programming software.
- (iii) Running the same 50 tests on the finished version every night.

(b) The team uses an IDE. Name the IDE tool that is helping in each situation. **[4]**

- (i) A programmer types `REPAET` and it changes to `REPEAT` by itself.
- (ii) After typing `PROC`, a list appears offering `PROCEDURE`.
- (iii) Keywords appear in bold blue and comments in grey.
- (iv) A message points to line 58 and says a closing bracket is missing.

(a)

- (i) Interpreter **B1**. It translates and runs the code a line at a time and halts at the first error in the changed procedure, so it can be tested before the rest of the program is finished **B1**.
- (ii) Compiler **B1**. It produces an executable file, so the office computers can run the program without any translator installed **B1**.
- (iii) Compiler **B1**. The program is translated once into an executable file, so the nightly runs don't need retranslating and run faster **B1**.

(b) (i) auto-correction **B1**; (ii) auto-completion **B1**; (iii) prettyprint **B1**; (iv) error diagnostics **B1**.

Common mistakes

- **Answering "what is meant by" with benefits.** Describe the language itself (English-like statements, needs translating, portable). "It is easier to read" is a benefit, and on its own it doesn't say what the language is.
- **Mixing up the two families.** Candidates often think "high-level" means harder, and tick "harder to debug" or "directly manipulates hardware" as a high-level benefit. High-level is nearer to people; low-level is nearer to the hardware.
- **Describing how when asked why or when.** A question about which translator to use while developing wants its **advantages** and the stage, not an account of how each one works. A question about the operation wants the process, not benefits.
- **Leaving out the order.** Saying only that a compiler "turns the program into machine code and starts it" hides the key point: no line runs until every line has been translated. For an interpreter, make clear that each line is carried out straight after its own translation, before the next one is touched.
- **"It shows the errors one by one."** That reads like a list of every error. Say the interpreter **halts** at the first error it meets and waits for the fix.
- **One-sided differences.** "An interpreter translates line by line" is only half a difference; add "whereas a compiler translates the whole program at once".
- **Saying translators debug.** Translators **find and report** errors; the programmer fixes them.
- **Naming a brand.** "Python" is not a type of language; write "high-level language".
- **Answering "binary"** when asked for another low-level language after machine code. Binary **is** machine code; the answer is assembly language.
- **Using the same translator twice** when the question says to choose a different one for each task.
- **Giving operating system jobs for an IDE,** or repeating the tool's name as its role.
- **Handing back what the question gave you,** such as "easy to read and write" when that is the example it asks you to go beyond.

How to get the marks

- **"Identify", "give", "state"** (1 mark each): the exact term, such as "assembler", "interpreter", "compiler", "machine code", "low-level language" or "executable file". A name, not a description.
- **"Describe"** (a meaning or how something works): one separate fact per mark, using the key words: English-like statements, mnemonics, machine independent, everything translated before running, line by line, halts at the error, report of all errors, executable file.
- **"Explain" a difference or a comparison:** both sides, joined with "whereas". Mark schemes usually cap the marks for one translator's side at 2.
- **"Explain why" or "when" a translator suits:** name the stage (development or the finished program) and give each point an effect with "so": "it halts at the faulty line, **so** debugging is easier".
- **Choosing a language for a setting:** the choice scores nothing on its own; the reasons score, and they must fit both your choice and the setting.

- **Tick boxes:** tick exactly as many as asked. In a table where a row may need more than one tick, check each column separately.
- **IDE tool + role:** a mark for the tool and a mark for what it does for the programmer.
- **Read what the question has already given you** (a translator, a reason, a tool) and give **different** ones.

Quick check

1. State whether each statement about low-level languages is true or false. (a) Machine code needs an assembler before the CPU can run it. (b) An assembly language program runs only on the type of processor it was written for. (c) A low-level program usually uses less memory than the same program written in a high-level language. (d) Each mnemonic in assembly language usually stands for several machine code instructions.
2. A 25-line high-level program has one error, in line 18. (a) How many lines have been executed when an interpreter reports the error? (b) How many lines have been executed when a compiler reports it? (c) After the error is fixed, which translator leaves a file that could be run again tomorrow without any translating?
3. Three files are waiting to be run: `lights.asm` (assembly language), `quiz.src` (high-level language) and `boot.bin` (machine code). For each file, name the translator needed before the CPU can run it, or write "none".
4. A friend says: "High-level languages are better than low-level languages in every way." Give **two** facts that show your friend is wrong.
5. Name the IDE tool you would use to: (a) type your code in and make changes to it; (b) run your program without leaving the IDE and watch what it outputs.

Answers

1. (a) False: machine code needs no translator; the assembler translates assembly language. (b) True. (c) True. (d) False: each mnemonic usually stands for one machine code instruction.
2. (a) 17 (lines 1 to 17 run, then it halts at line 18). (b) 0: a compiler translates everything first and runs nothing while there are errors. (c) The compiler, because it produces an executable file.
3. `lights.asm`: an assembler. `quiz.src`: a compiler or an interpreter. `boot.bin`: none; the CPU runs machine code directly.
4. Any two: its translated programs usually run more slowly and use more memory; it has to be translated before it can run, which takes time; it can't drive the hardware directly or use instructions only one processor has.
5. (a) The code editor. (b) The run-time environment.

Past questions to try

- [0478/13/M/J/23 Q4](#): a high-level language benefit, then naming the translator for each statement.
- [0478/12/F/M/23 Q2](#): choosing a high-level or low-level language for a game, then how a compiler and an interpreter work.
- [0478/11/O/N/24 Q6](#): assembly language for an embedded system, and three ways an IDE helps.
- [0478/11/M/J/22 Q6](#): a similarity and two differences between a compiler and an interpreter.