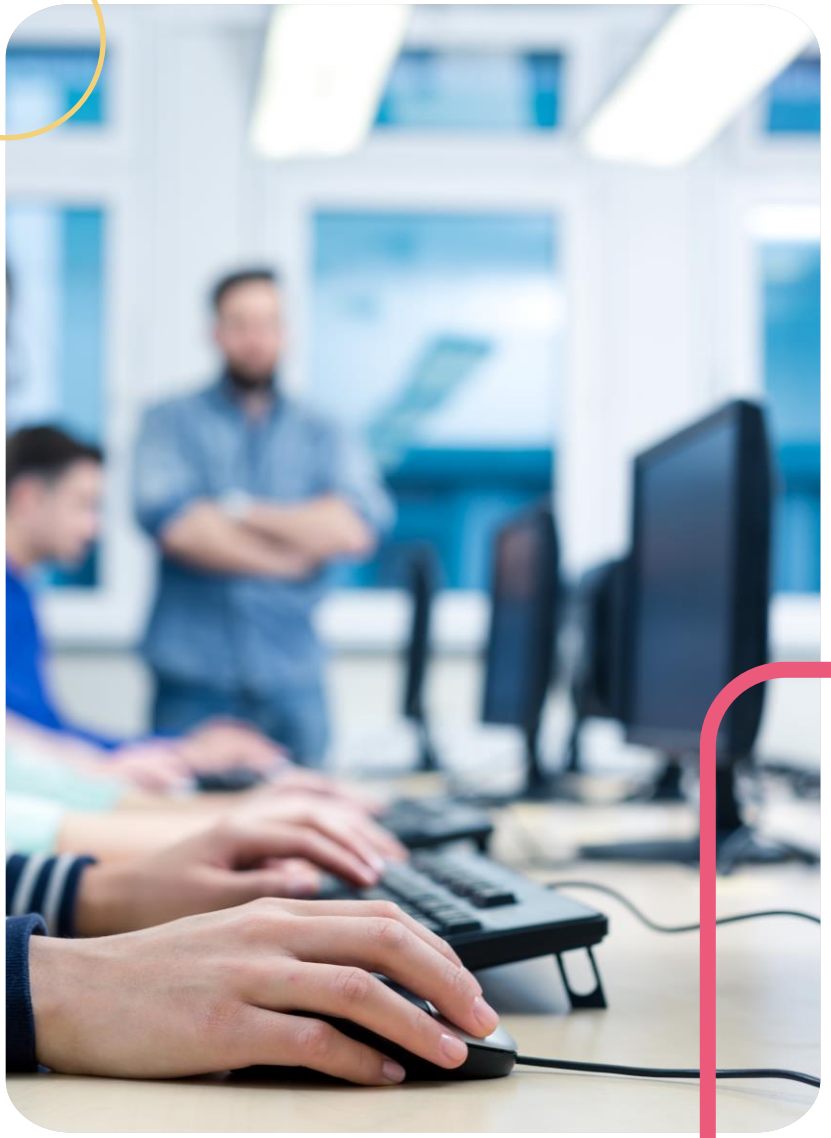
Diploma in

Computer Science

Programming Languages



Objectives

- 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

Challenge

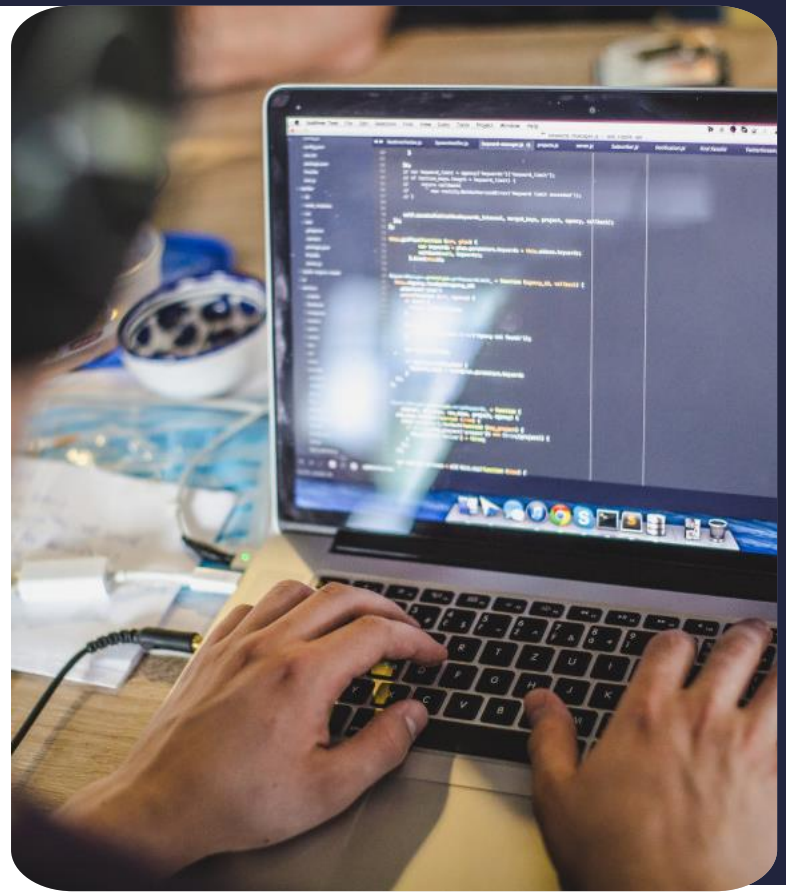
To which programming language generation do the tools used to code Siri belong?

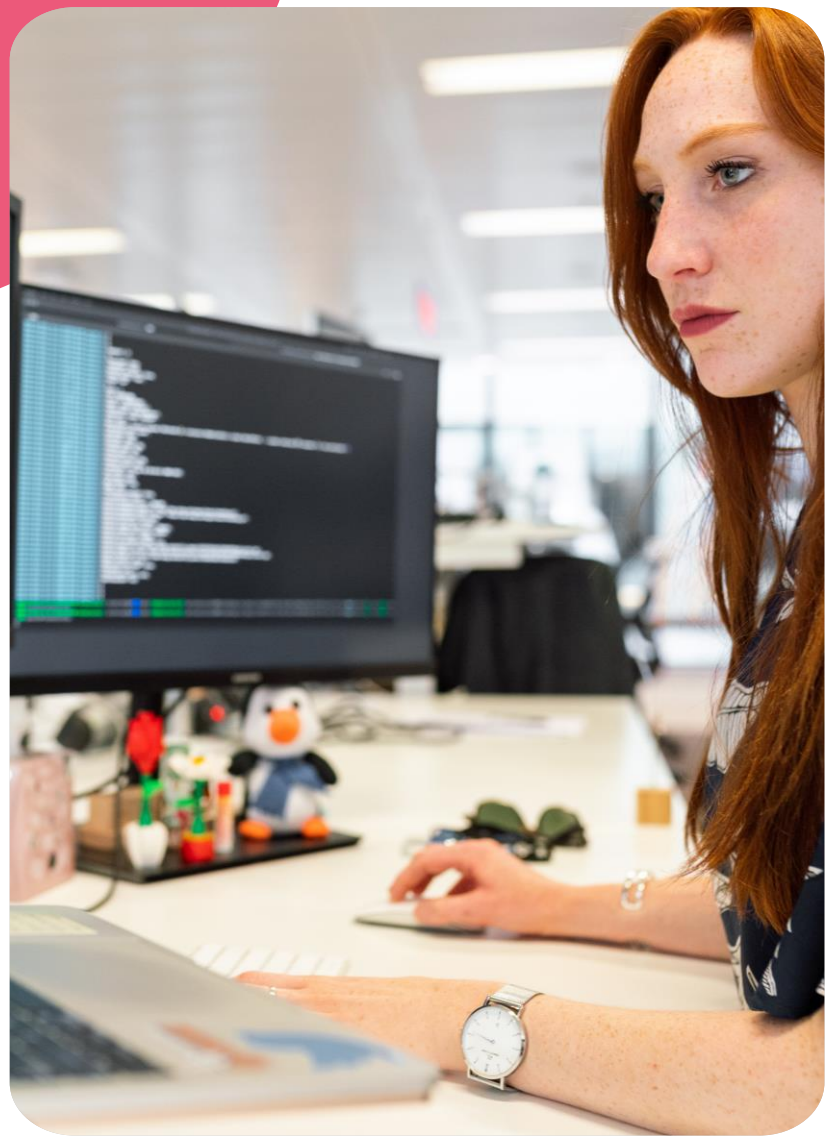
#PLGSiri

History of programming languages

History of programming language

- Over 200 years back to Jacquard machine
- Programming languages evolved along with computers
- First computer programmer Ada Lovelace

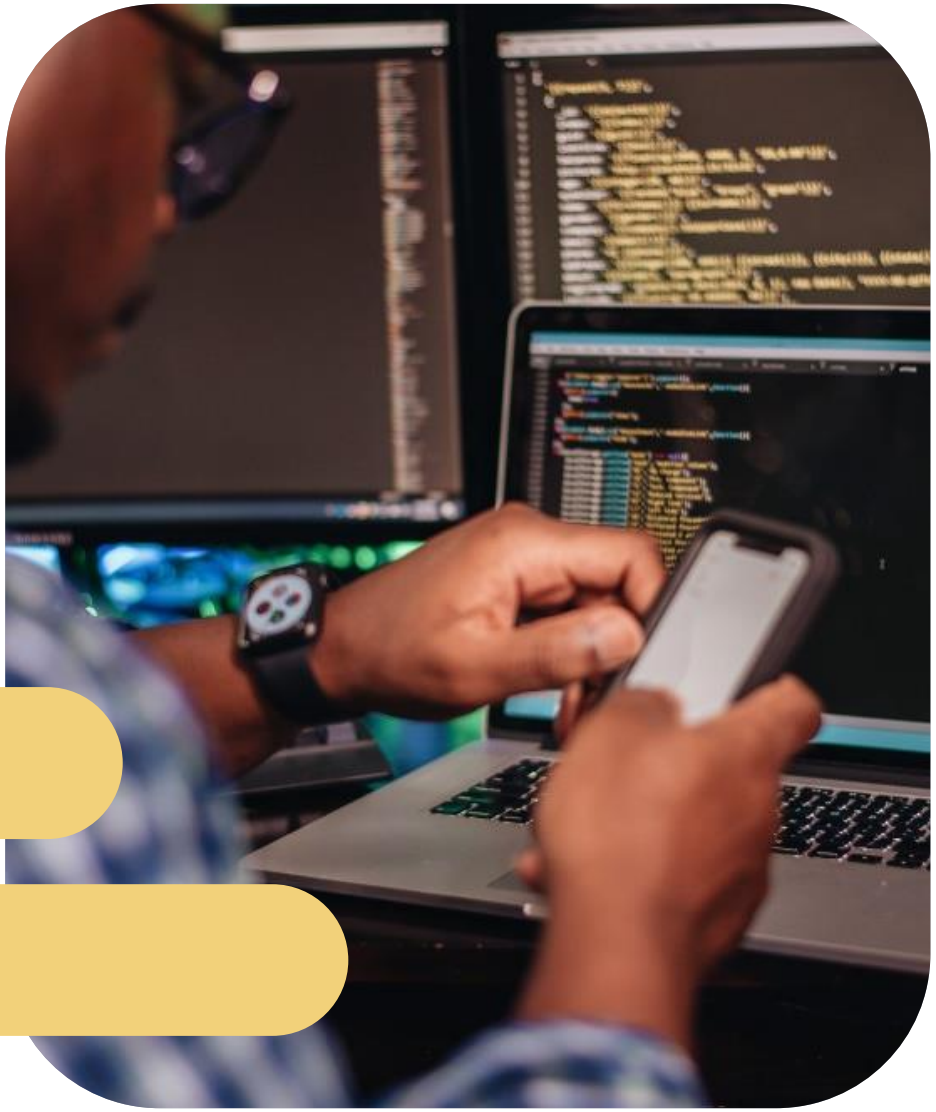




- Development of programming languages tied in with first stored-program computers
- Late 1940s - chunks of machine code were converted into words like ADD, SUB and MULT
- Kathleen Booth invented assembly language

- 60s – 70s: many new programming languages invented, consolidated and elaborated
- Focus was on programming large scale systems using modules
- Internet Age – focus switched to programmer productivity and the development of scripting languages

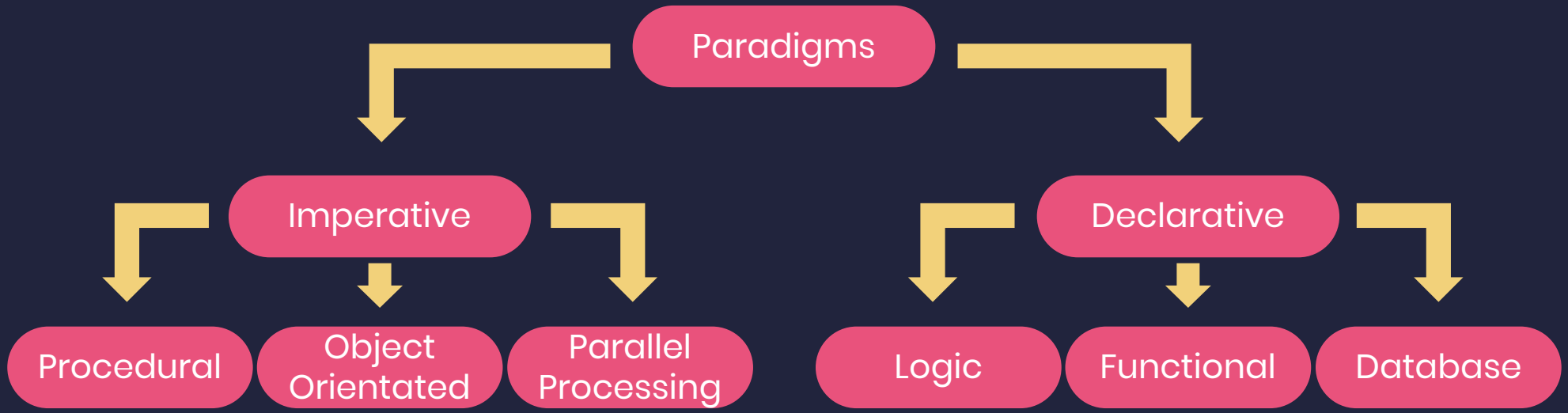
Types of programming languages

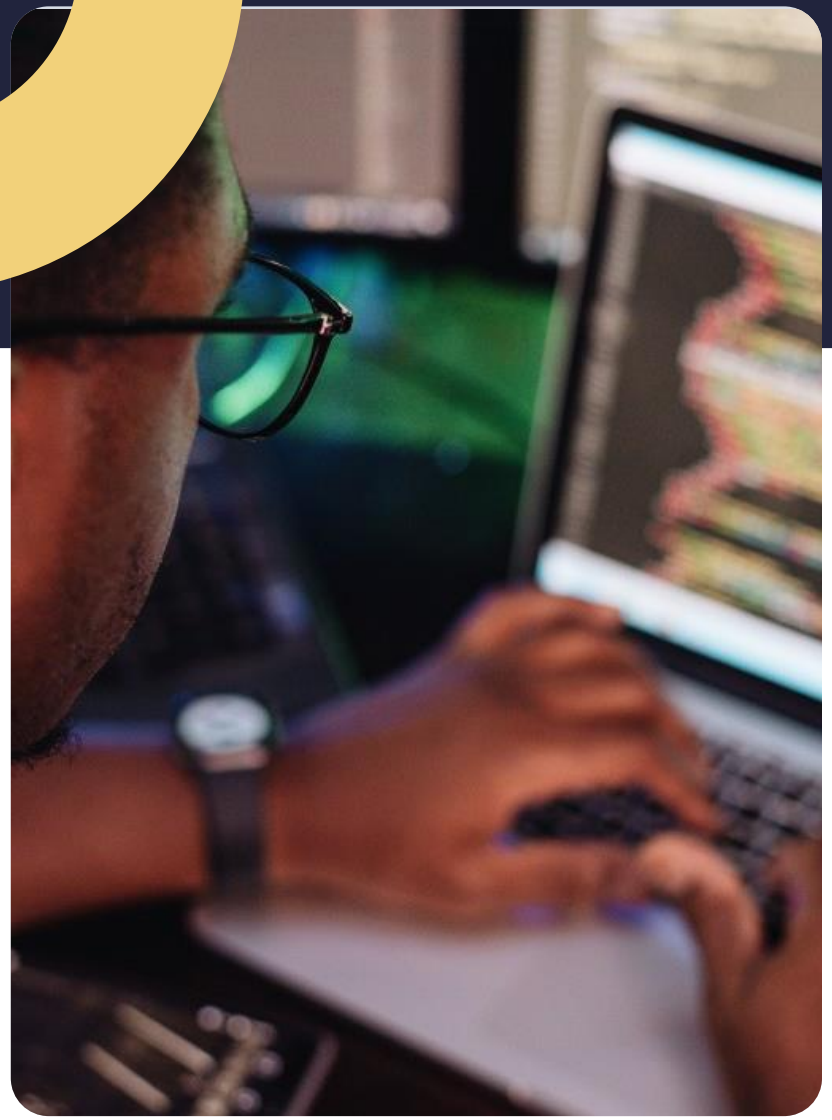


Types of programming languages

- Many types of programming languages classified according to their level of difficulty
- A programming language - a systematic method of describing a computational process
- Programming languages now general and all-purpose – classified according to qualities and use cases

Programming language paradigms

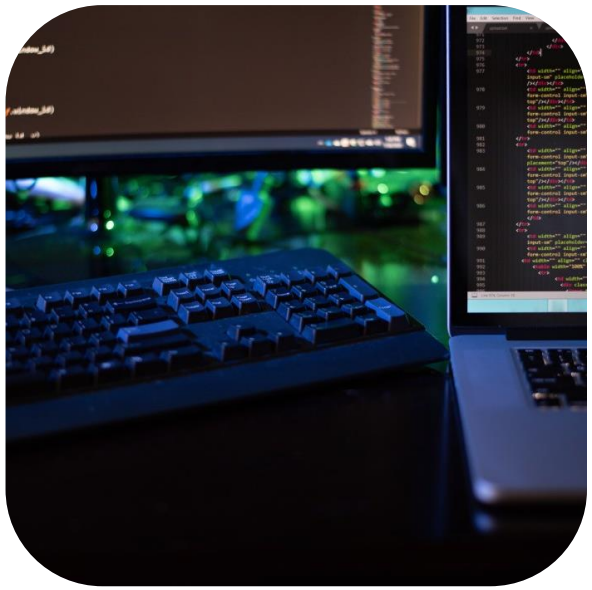




Imperative programming language

- Close relationship with machine architecture
- Works by changing the program state through assignment statements
- Performs a task step by step
- Simple to implement but less efficient and unable to solve complex problems
- Parallel computing not possible

Imperative programming paradigm branches



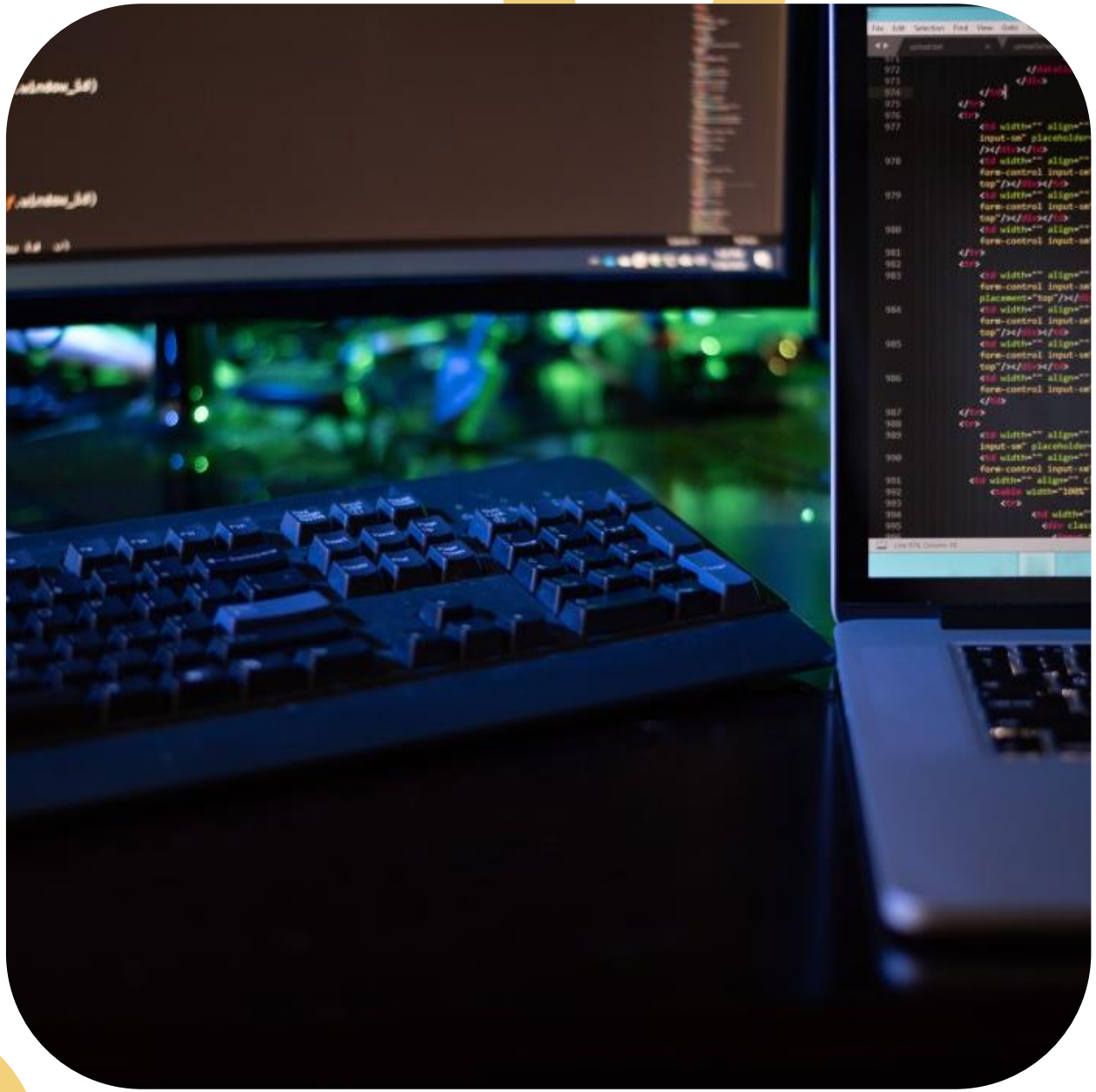
Procedural programming

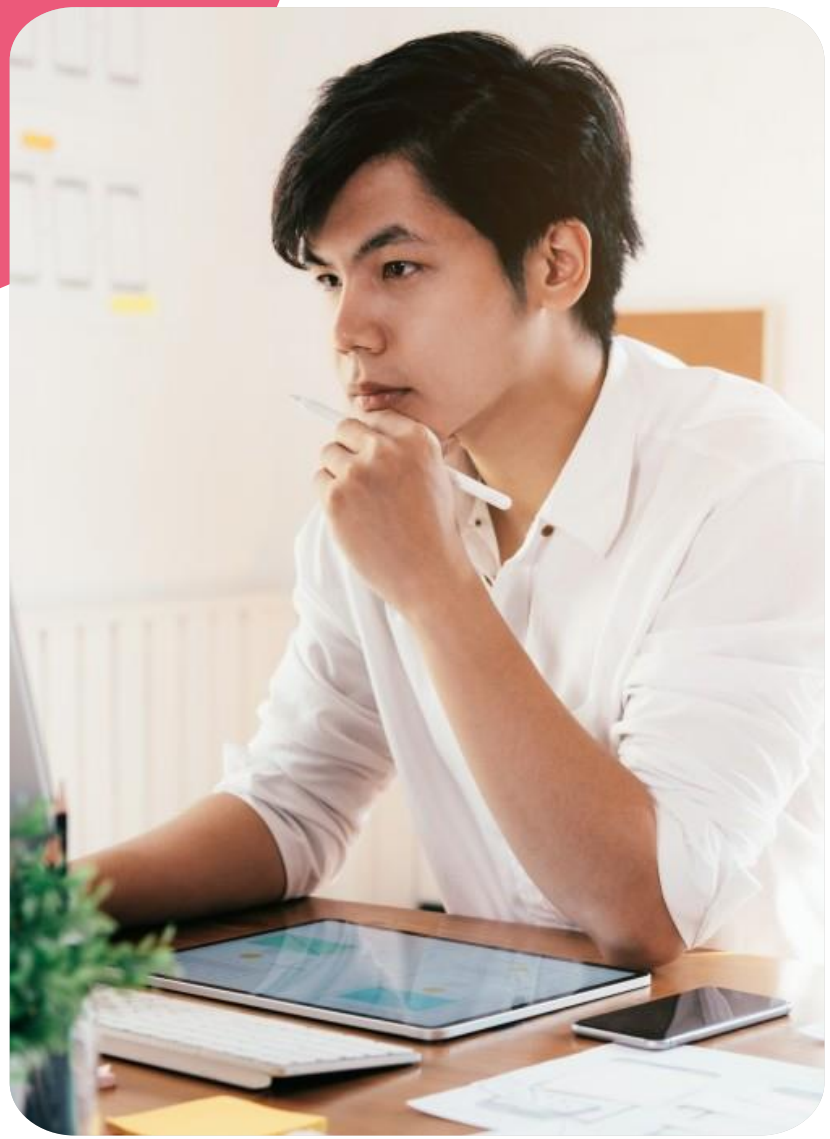
Object-oriented programming

Parallel-processing programming

Procedural programming

- Executes a sequence of instructions that lead to a result
- Includes multiple variables and heavy loops



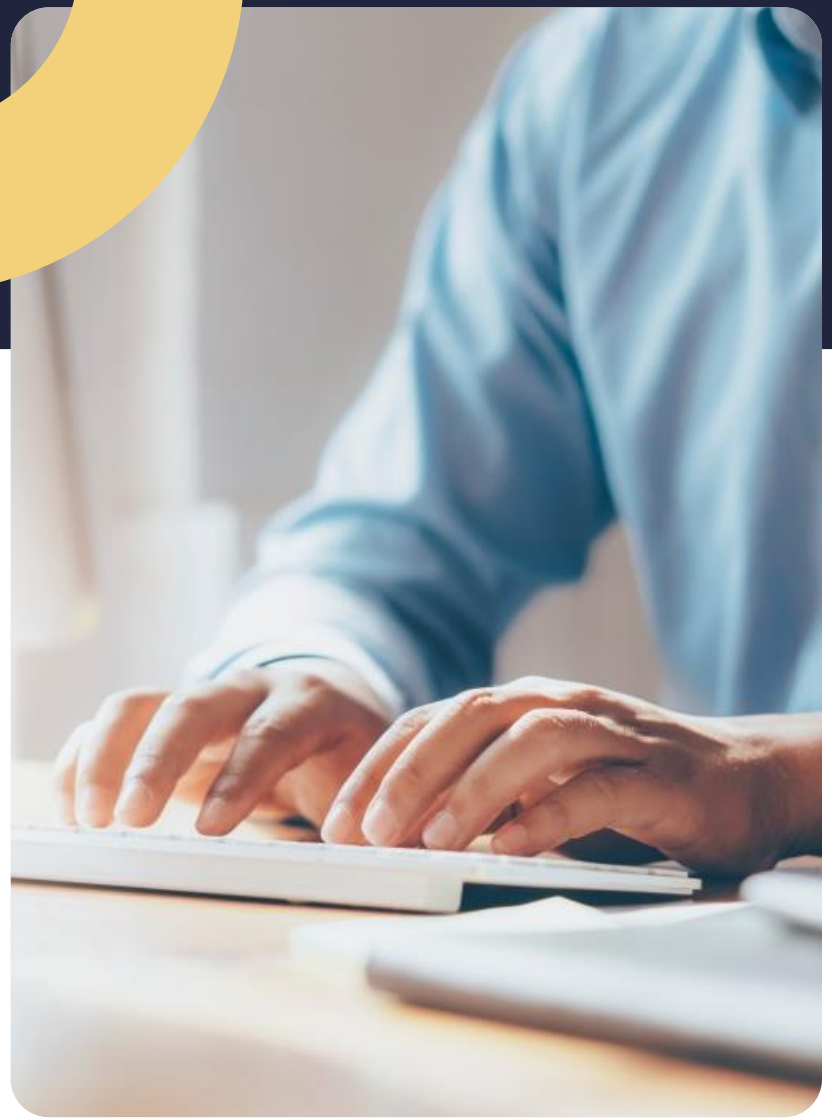


Object-oriented programming

- Views a problem in parts and solves each independently then interconnects the parts
- Reusability by using inheritance and polymorphism

Parallel processing

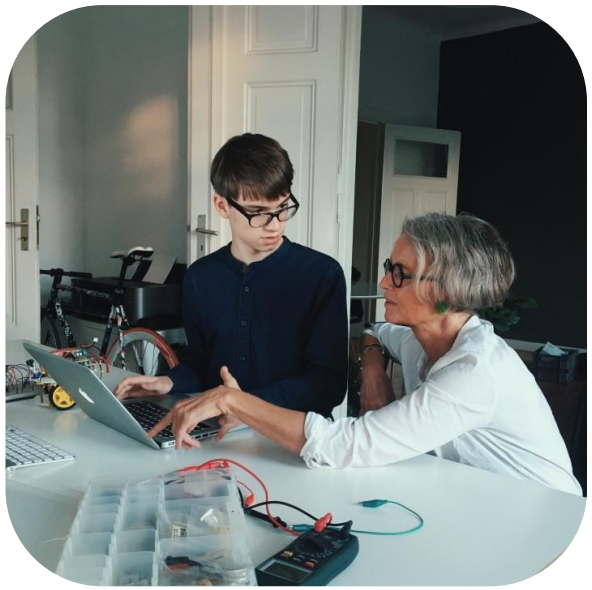
- Speeds up processing by distributing instructions among multiple processors
- Divide and conquer



Declarative programming language

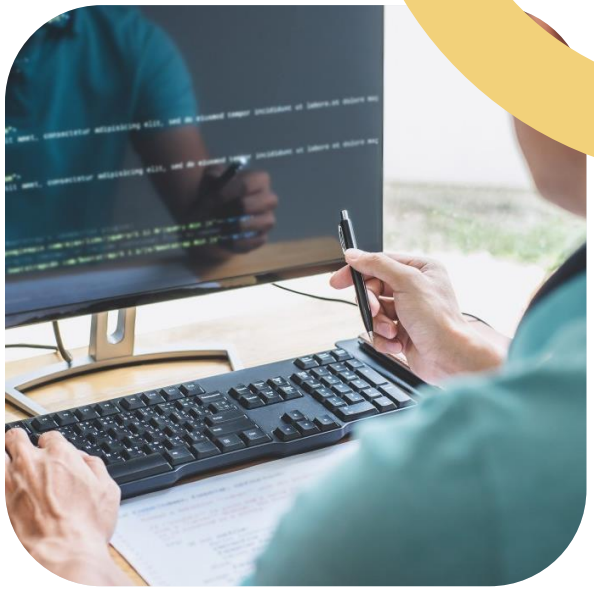
- Expresses the program in implicit terms
- Focus is **what** needs to be done rather than **how** to do it
- No loops, assignments, etc.
- Little difference between specification of a program and how it is implemented
- High level Instructions

Declarative programming paradigm branches



Logic programming

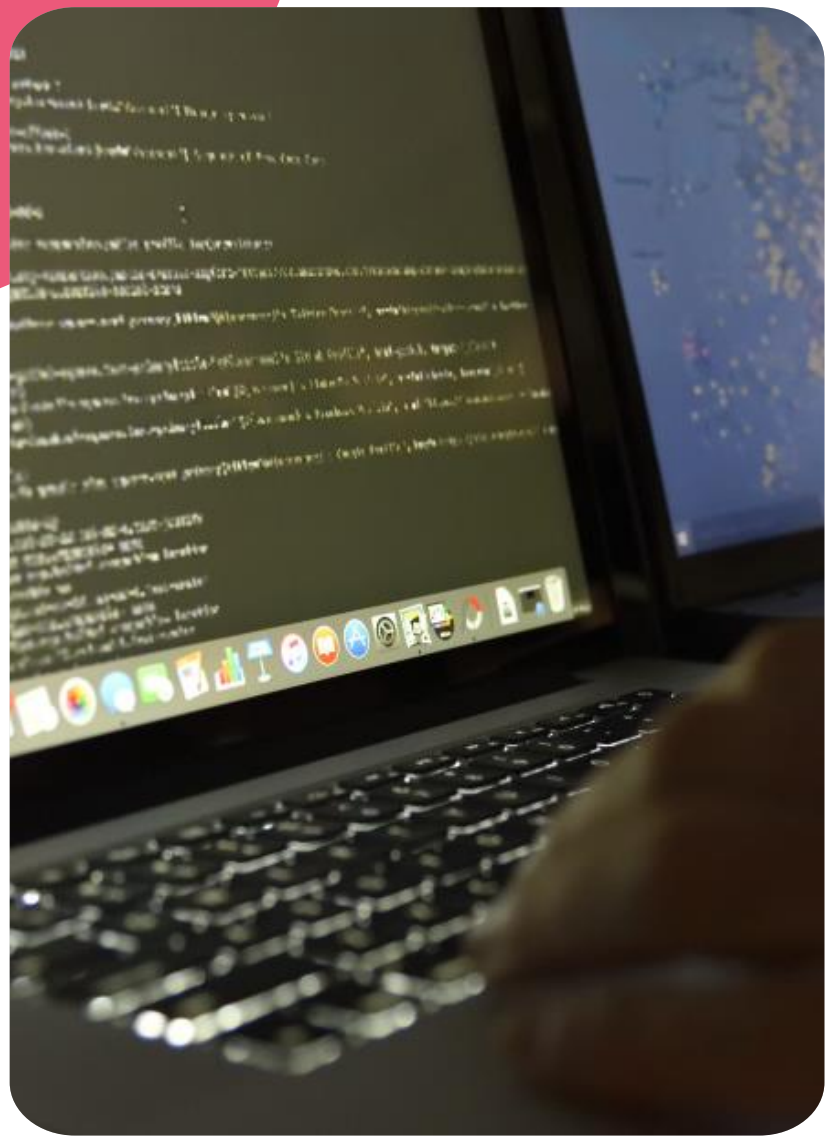
Functional programming



Database programming

Logic programming

- Declarative statements that allow the machine to reason about the consequences
- Does not tell computer what to do

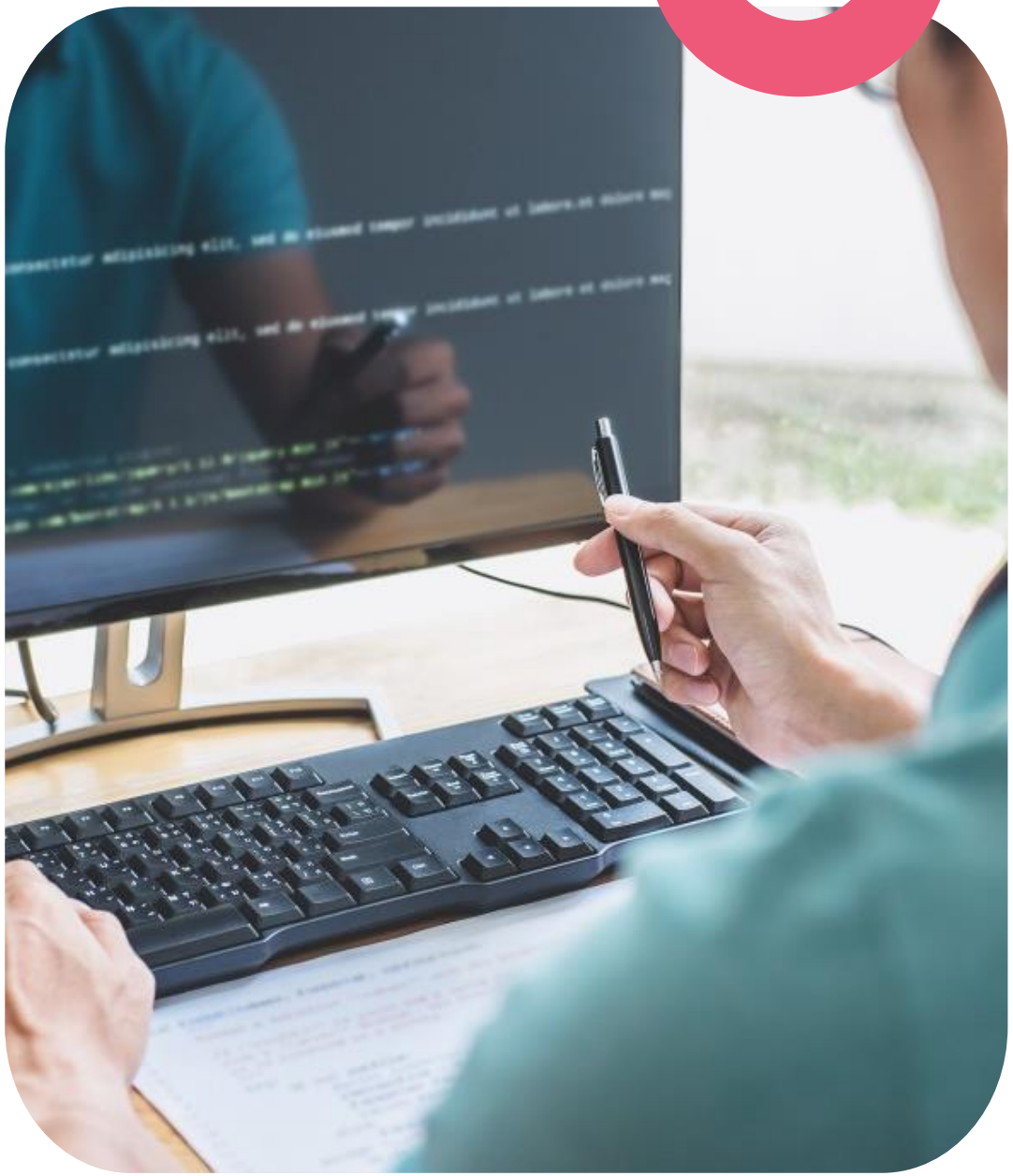


Functional programming

- Languages typically use stored data
- Primary focus is pure mathematical functions and immutable data
- All about the flow of the program – passing functions to functions and returning functions from functions
- Code is shorter, less error prone and easier to correct

Database programming

- Program statements defined by data rather than creating a series of steps
- Databases use Structured Query Language (SQL) for writing and querying data
- Can handle massive amounts of information
- Easy to maintain data consistency



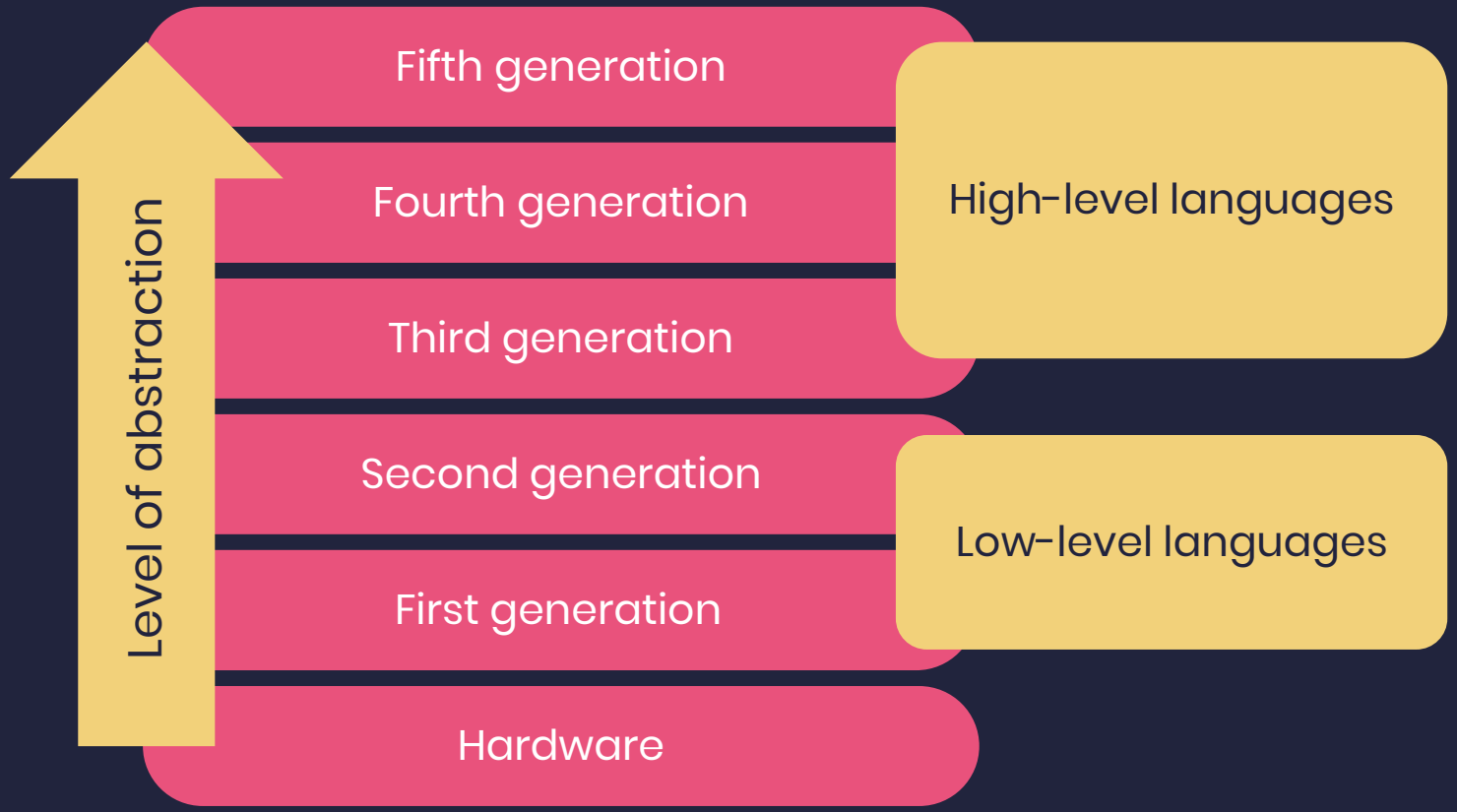
A final word on paradigm languages...

- Very few languages implement a paradigm 100%
- Many facilitate programming in one or more paradigms
- Multi-paradigm language used
- If accidentally supports multiple paradigms, referred to as single paradigm language

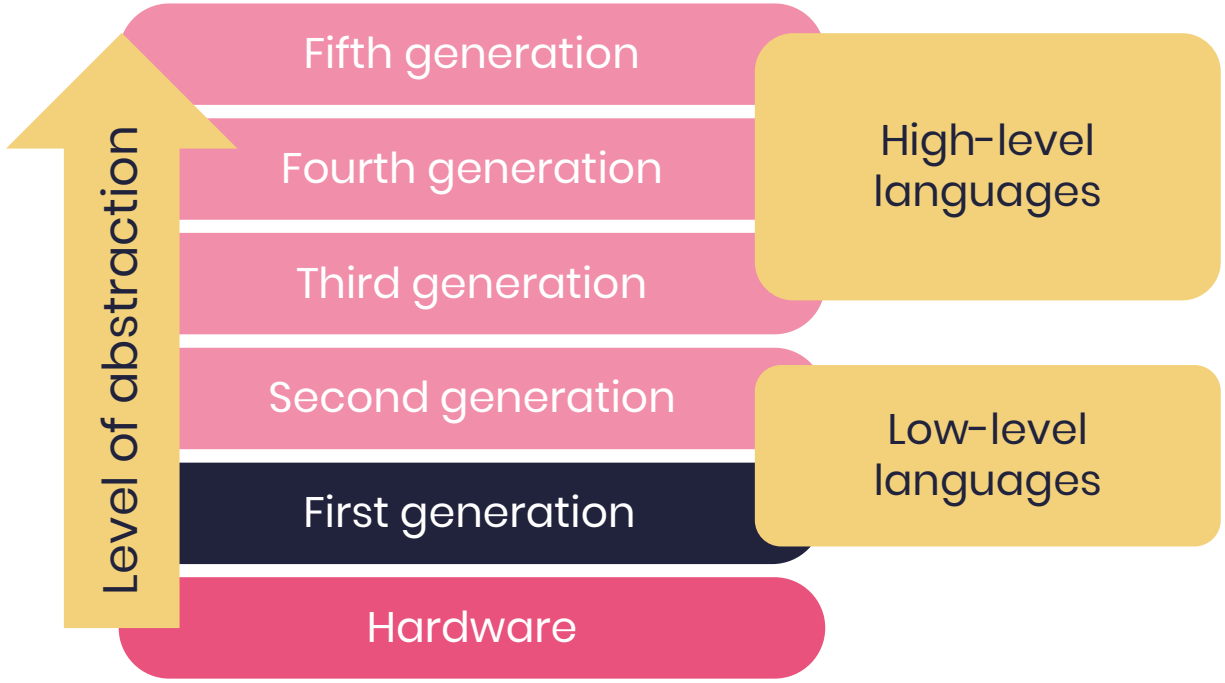
Did you know?

There are over 700 different programming languages - from C, JavaScript to Python, and hundreds more.

Generations of programming languages



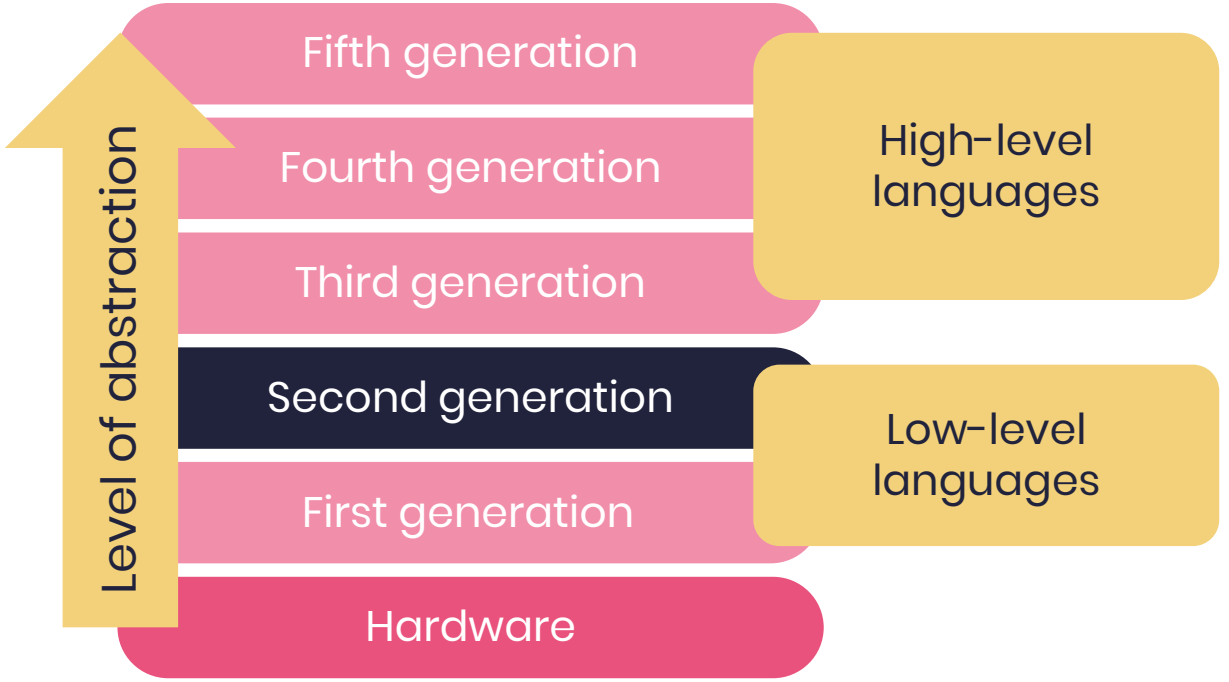
First generation language (1GLs)



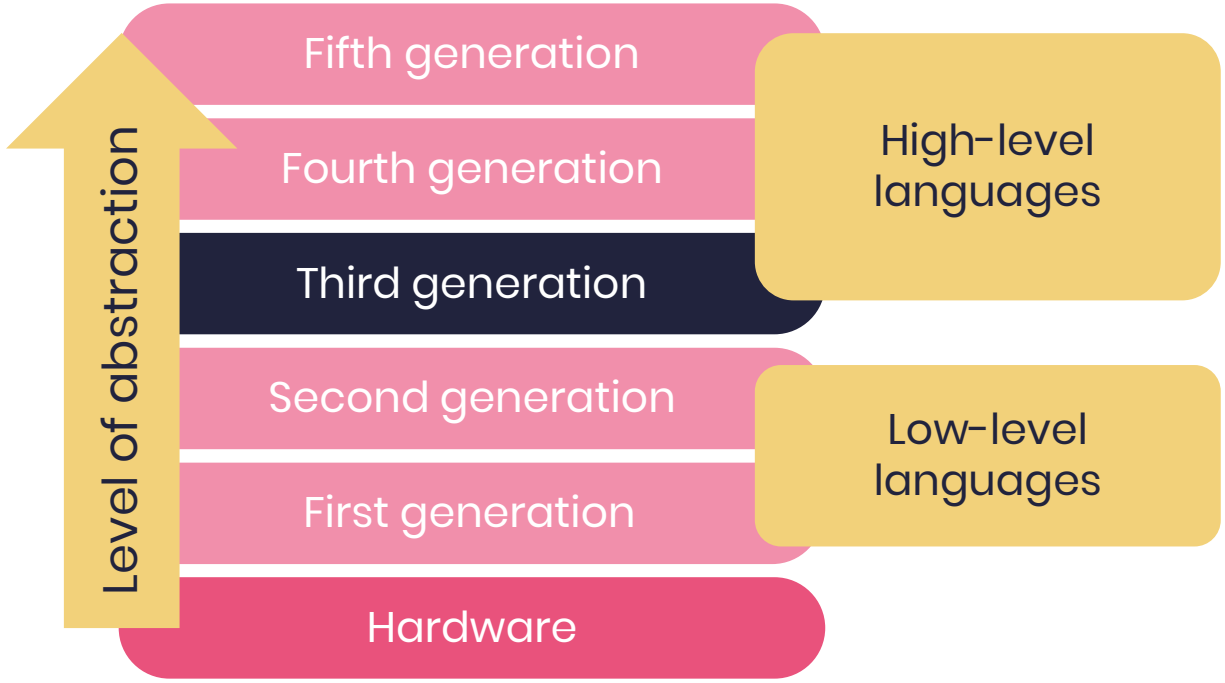
- Sequence of 1s and 0s
- Required meticulous attention to detail
- No need for translation
- Execution of these programs speedy and efficient
- Machine code is machine specific

Second generation languages (2GLs)

- Use mnemonics in place of 1s and 0s to represent opcode and operands
- Easier and less prone to errors
- Machine cannot directly read this code
- Has to be dismantled into 1s and 0s by a program known as an assembler



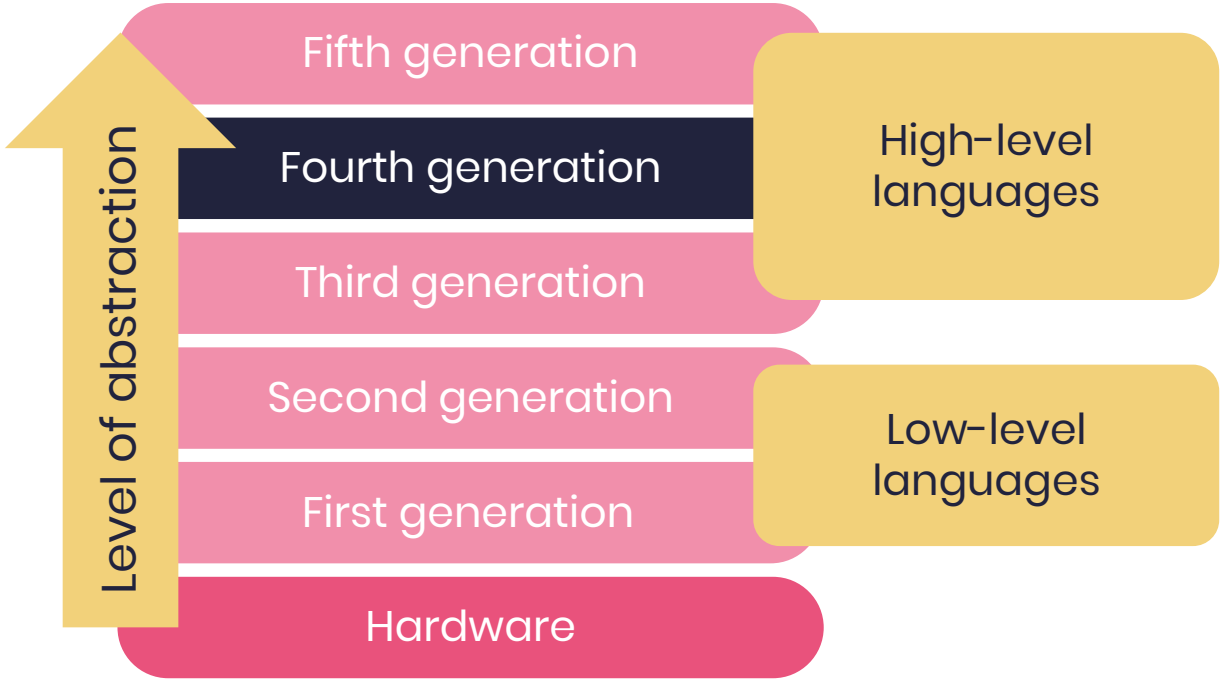
Third generation languages (3GLs)



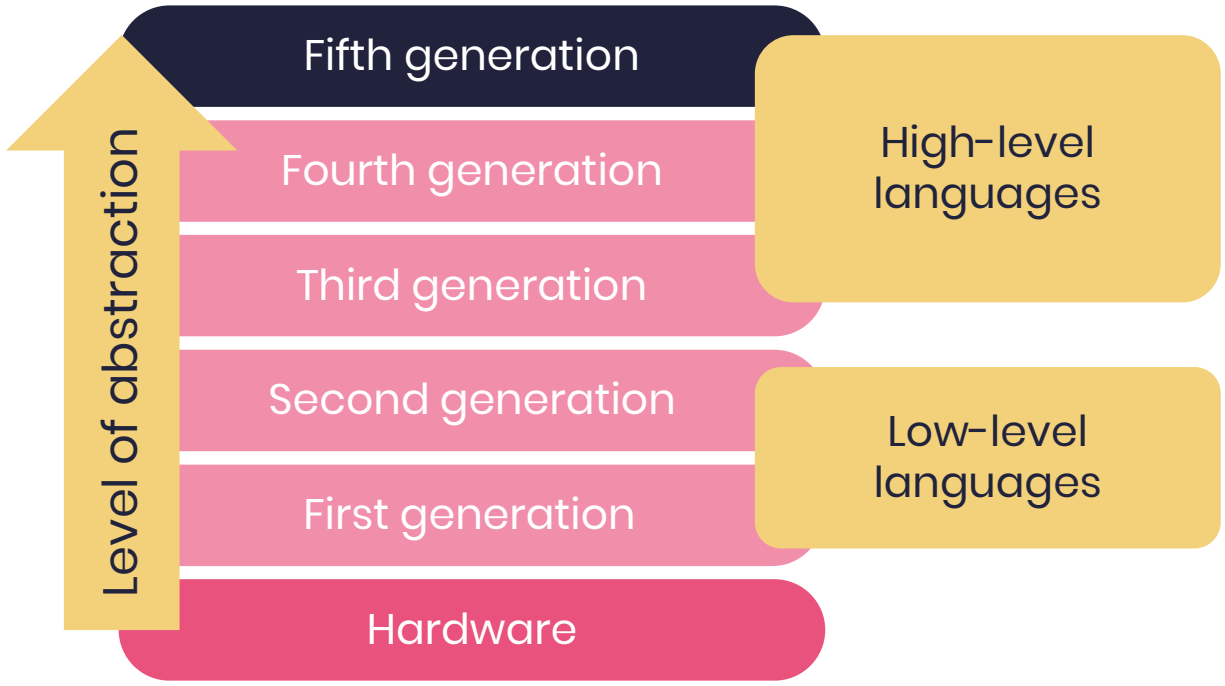
- Significant improvements compared to second generation languages
- Moved away from variables, data types and constants – made it easier
- Allowed for defining sections of code as subroutines
- Generate a source code that is converted by a compiler into object code then linked to produce machine code
- Can run on many different types of computers

Fourth generation languages (4GLs)

- Attempt to get closer to human language to reduce time, cost and effort
- Key is use of icons, graphical interfaces, and symbolic representations
- Mostly associated with databases and data processing



Fifth generation languages (5GLs)



- Build an application using constraints
- Programming language used to describe the properties of the logic of the solution
- Computer is free to search for a solution that conforms to specific constraints
- Attempt to get the computer to solve a problem without the programmer

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

- Wrong choice of programming language may produce bloated, inefficient software
- Costs time, effort and money
- Programming paradigms help reduce the complexity of programs
- Each paradigm has a use case

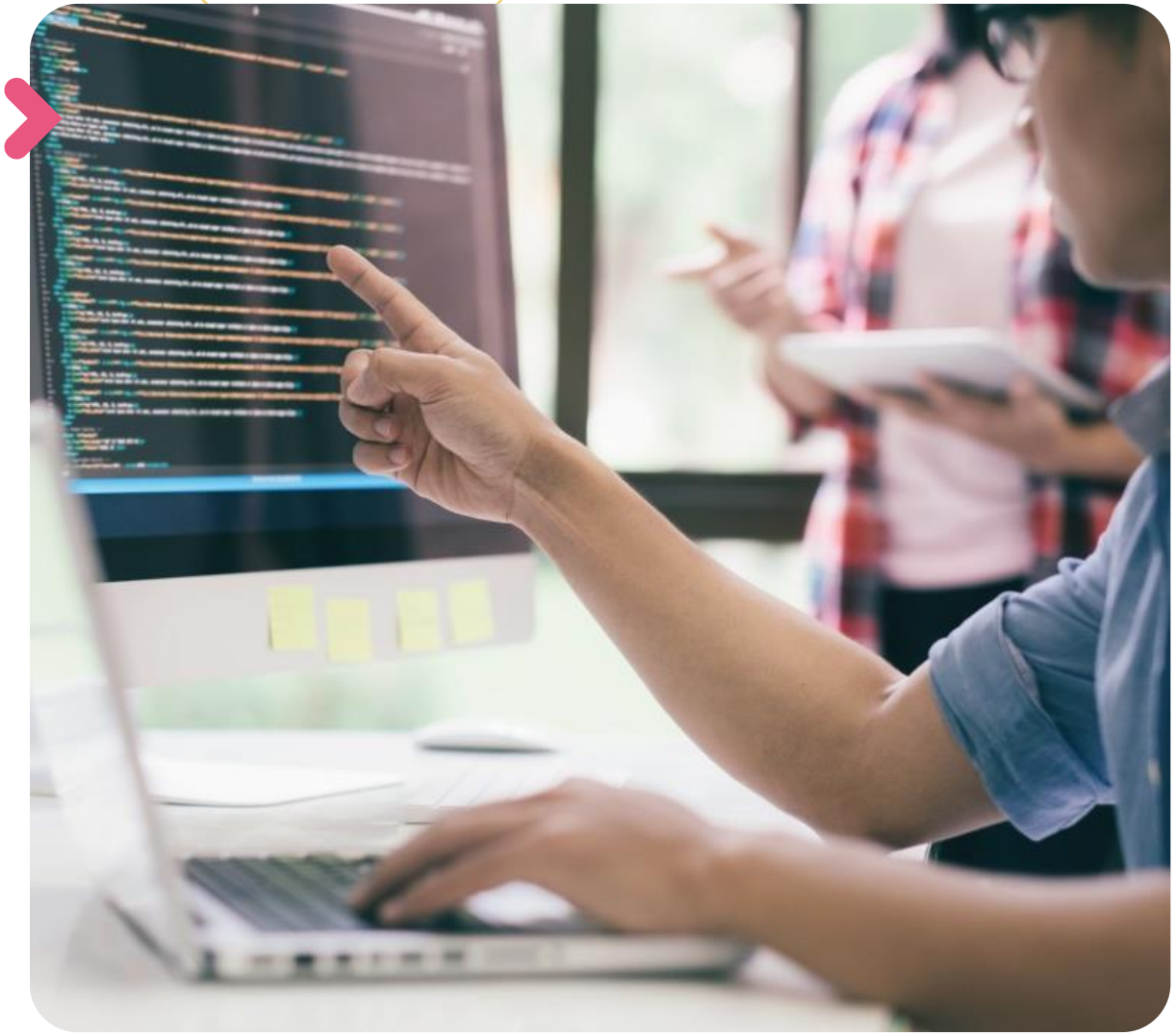
Questions

- 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?

Language levels

Language levels

- Low-level languages
- High-level languages



Low-level languages

- First to be developed
- Closely associated with the hardware they run on
- Not portable

Two types of low-level programming languages

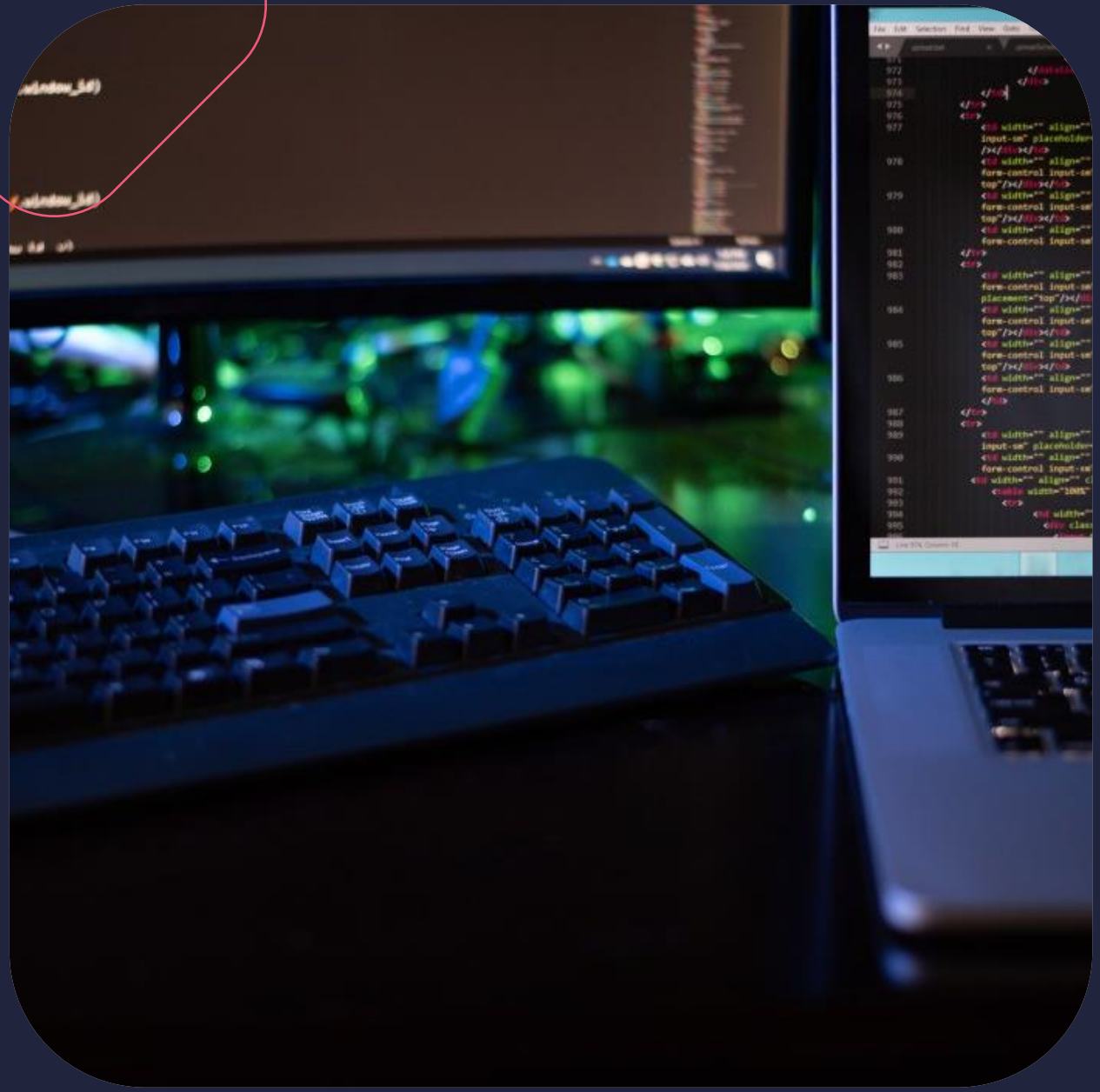
Machine language

Assembly language

Structure and function of low-level languages

- Machine language is directly run by the machine
- Fast execution and efficient use of hardware
- Assembler translates assembly language to machine language
- Require extensive knowledge of hardware and configuration

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
0000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```



Use of low-level languages

- Overall memory footprint is small
- Suitable for systems with severe memory constraints
- Used to develop specialist code
- Little abstraction of the hardware

High-level languages

- From the 3rd generation onwards
- Aim to make the programmer’s life easier
- Offer more hardware

Structure and function of high-level languages

- Created with the programmer in mind
- Focus on the problem being solved
- English-like statements - people had more freedom to be creative with their code
- Translation software required to convert the program into machine code

Did you know?

The first computer bug that was discovered in 1947 was a literal bug; a moth stuck inside Grace Hopper’s Harvard Mark II computer.

Uses of high-level language

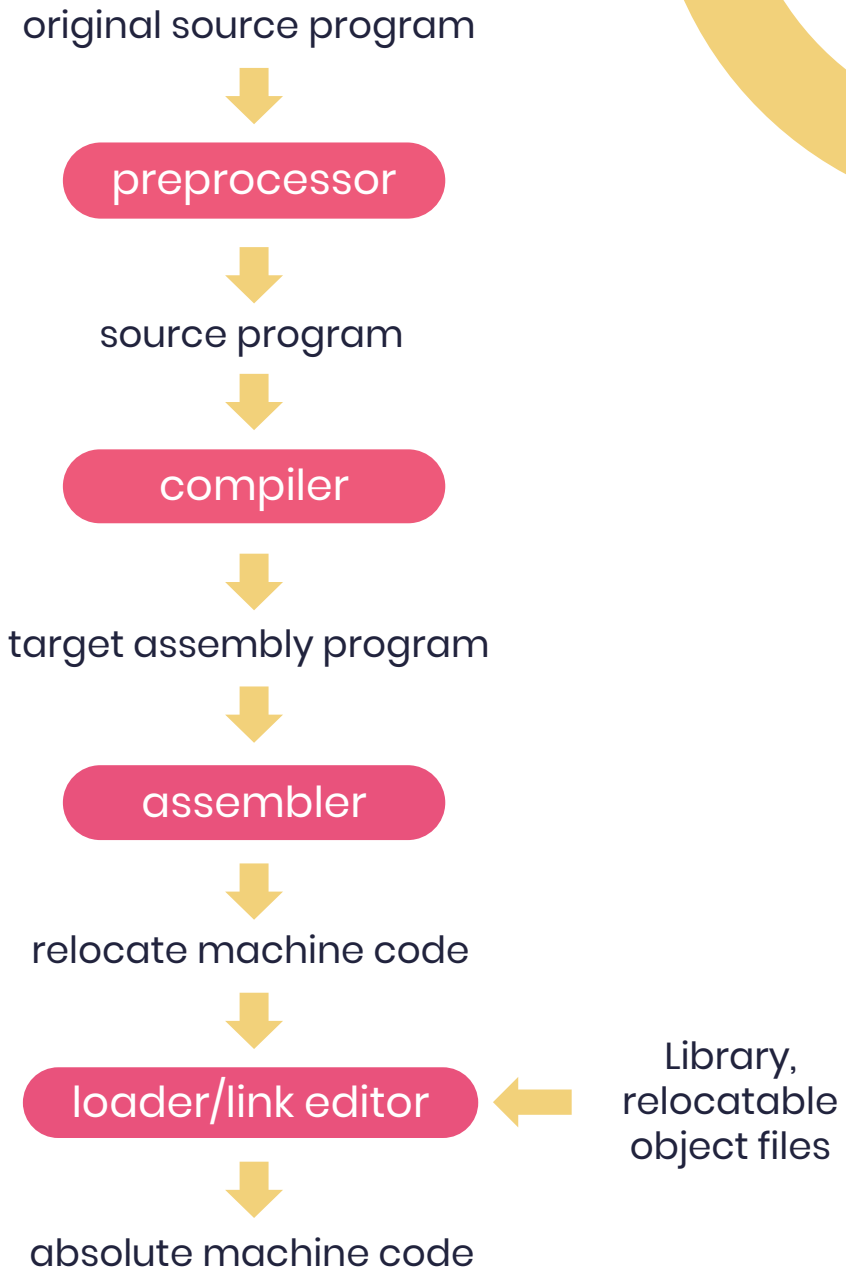
- Can run on machines of different architecture
- Code compiled for a specific system.
- Have huge libraries and large sets of keywords

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 process

A closer look at the compilation process

- Relies on 3 types of software: assemblers, compilers, and interpreters



Compilers

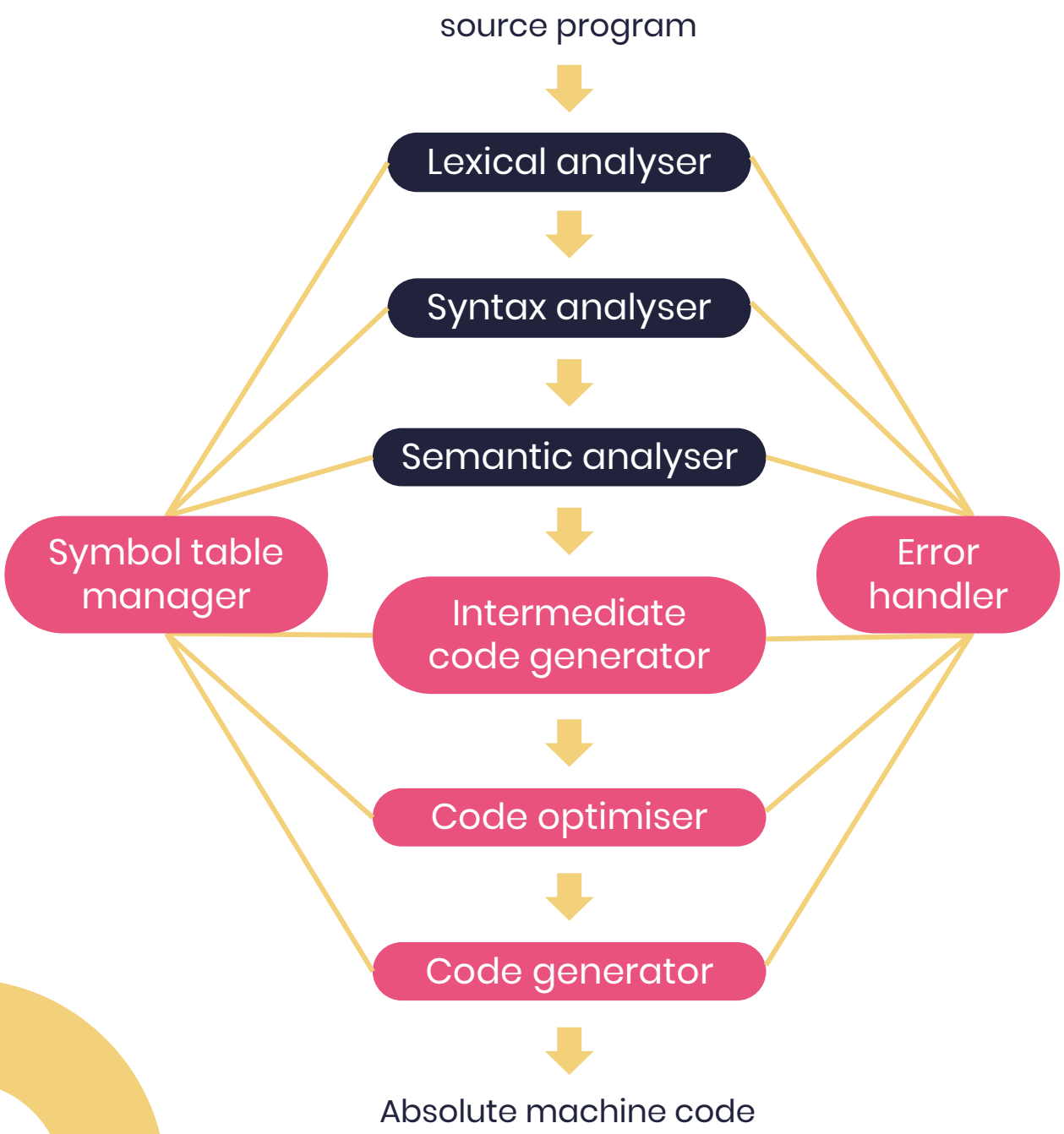
- Convert high-level languages into machine code
- Every IDE has a compiler
- Produce an executable

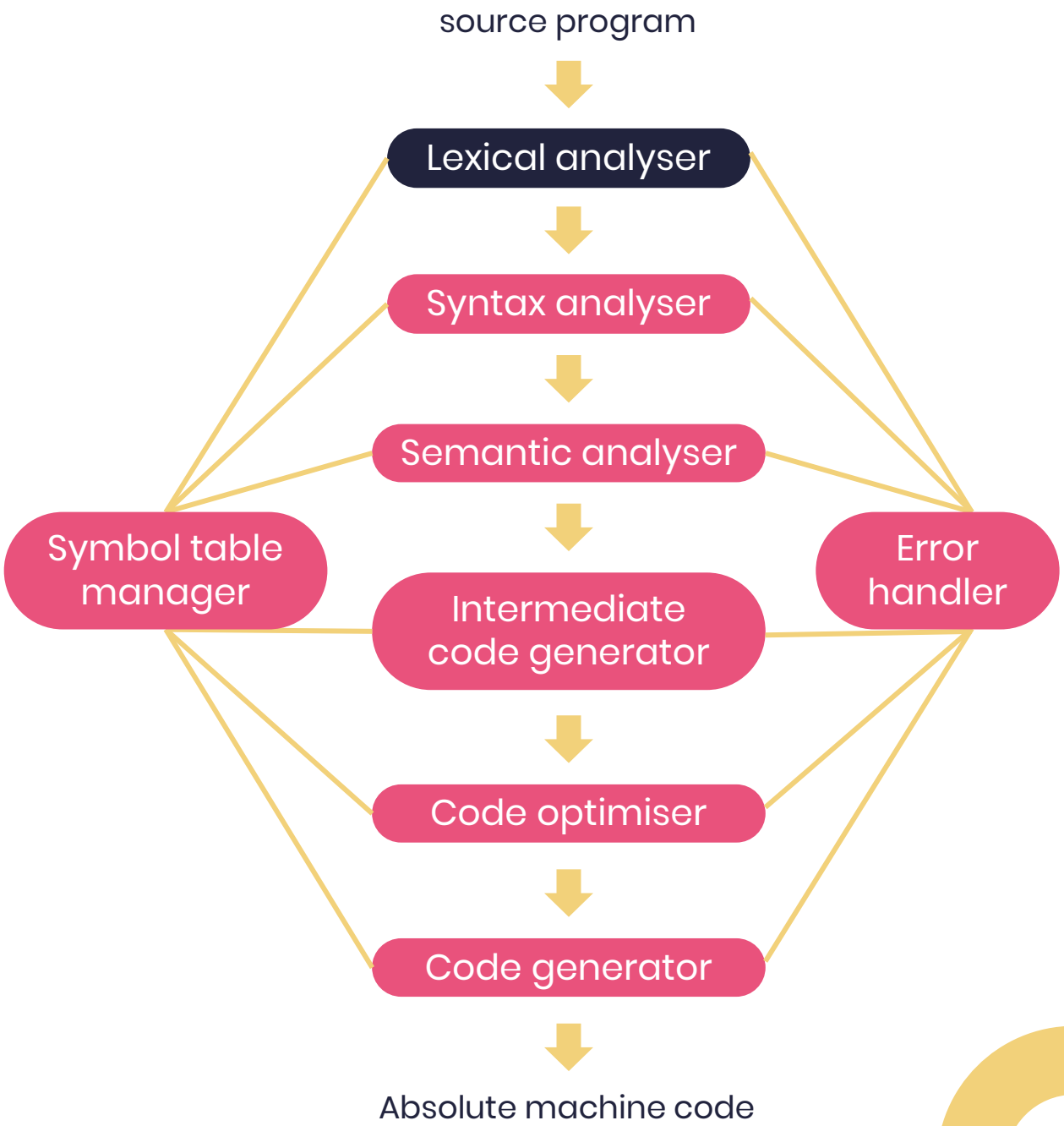
Interpreters

- Also translates code into machine readable format
- Compilers translate the whole program, interpreters translate the program line by line

The compilation process

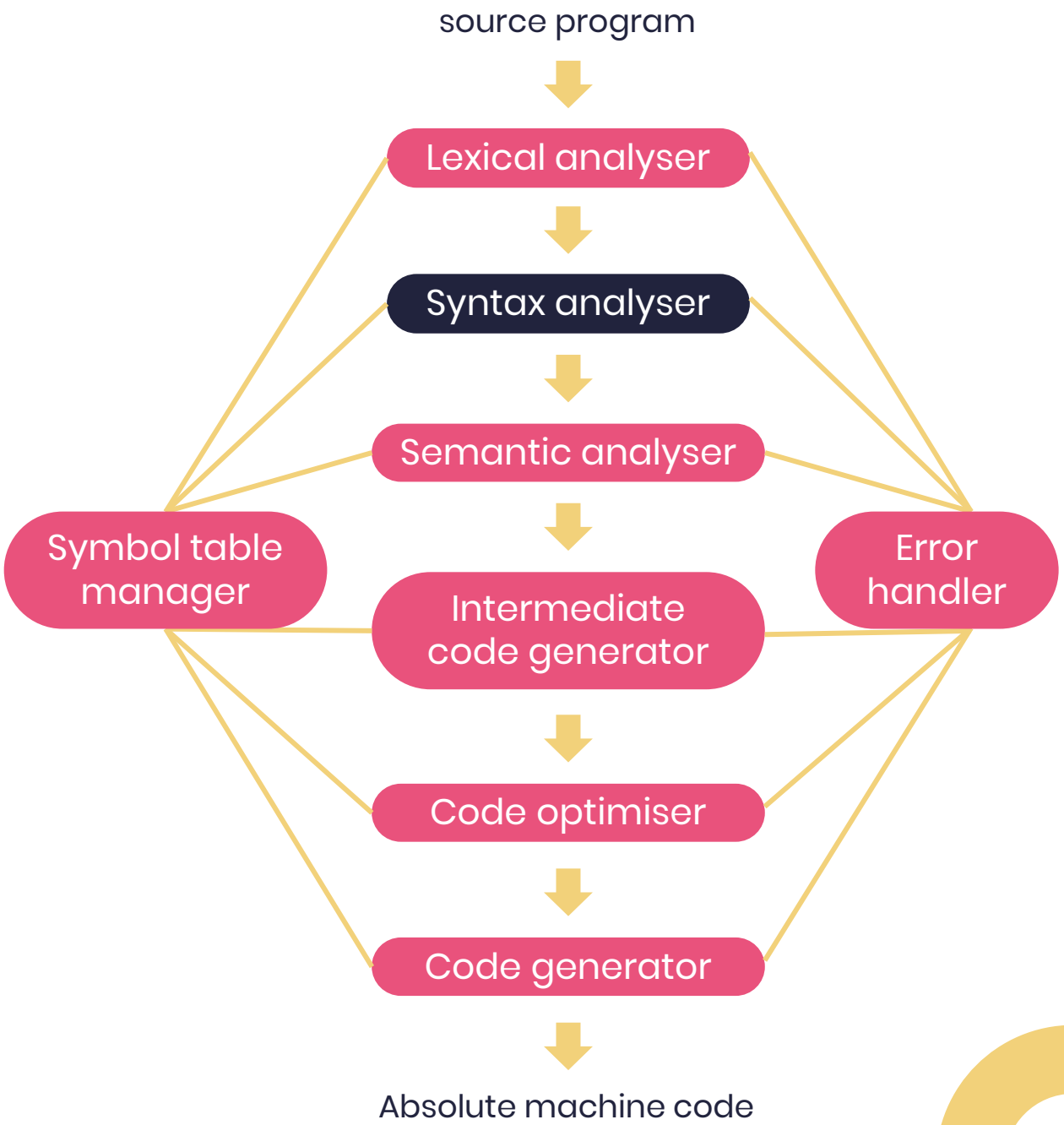
- Compilation and interpretation follow fairly complex processes
- Compilation process uses the abstract machine models
- Analysis comprises three stages: lexical analysis, syntax analysis and semantic analysis





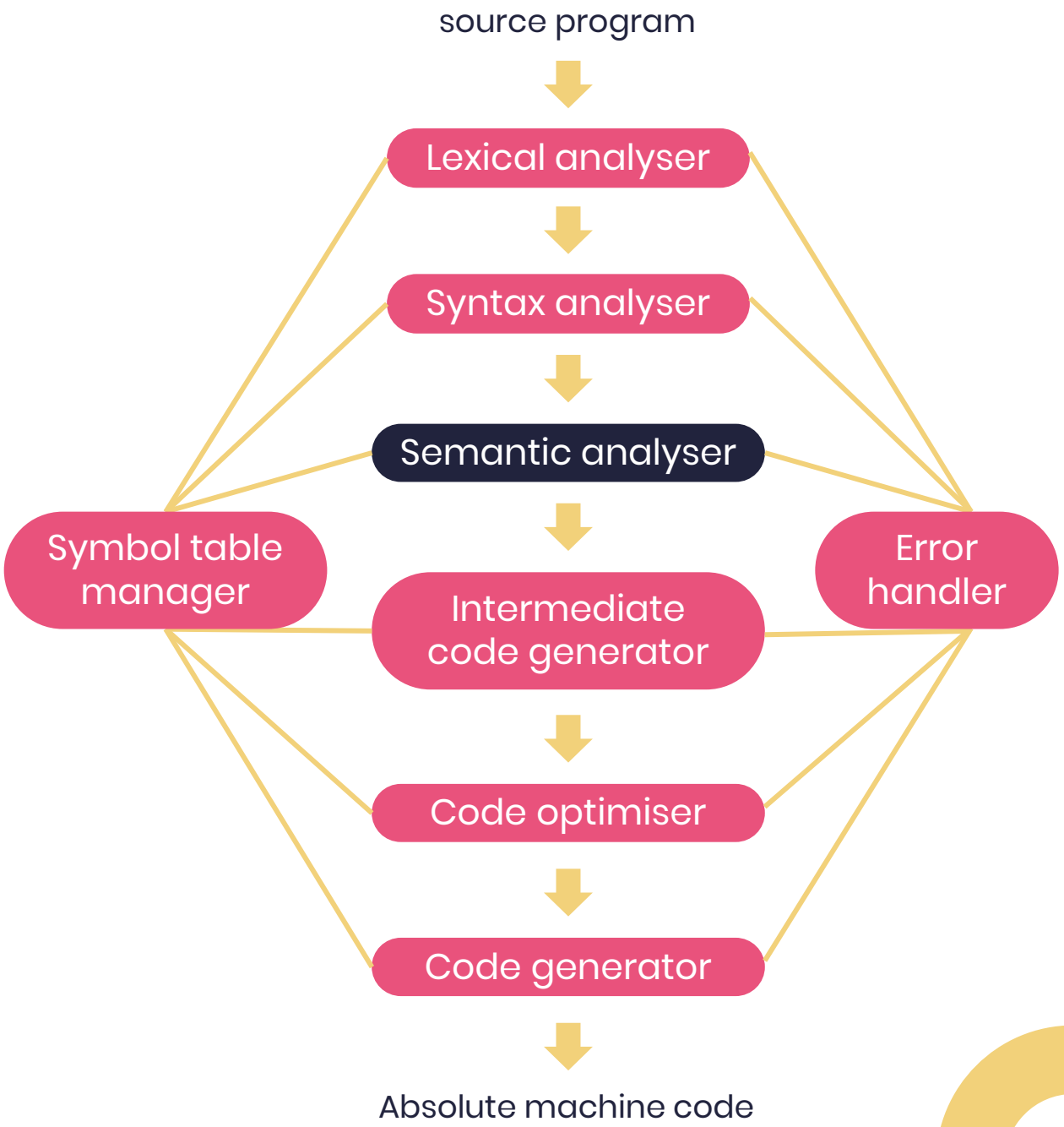
Lexical analysis

- Scanner reads lines of code and breaks them down into tokens, removing white space and other unnecessary things
- If scanner encounters an unrecognised symbol, it generates an error
- Every time a token is created, the symbols are stored in the symbol table



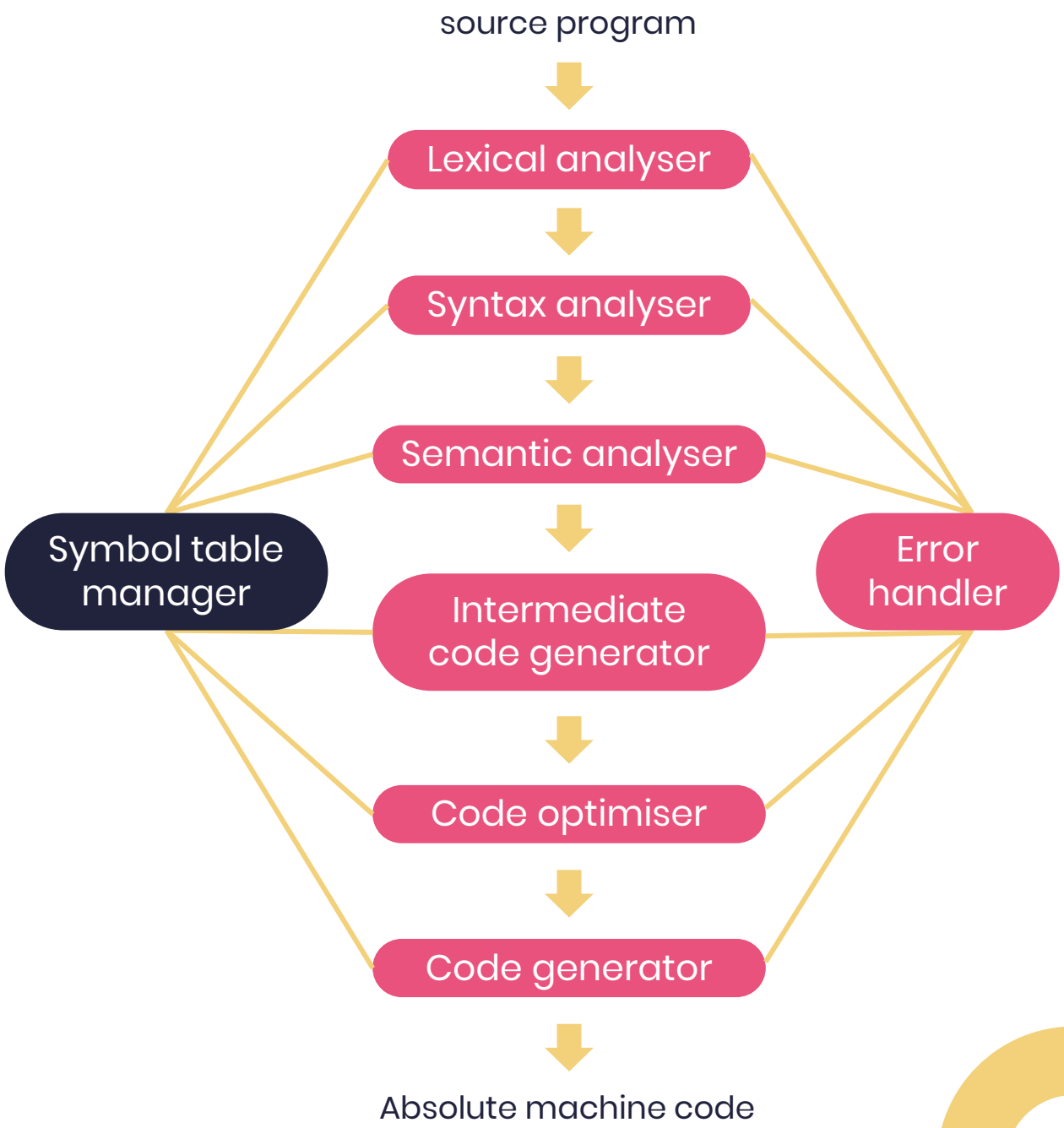
Syntax analysis

- Check code against the rules of the language
- If violations, it will pass these to the error handler
- No errors, the code is passed over to the semantic analyser



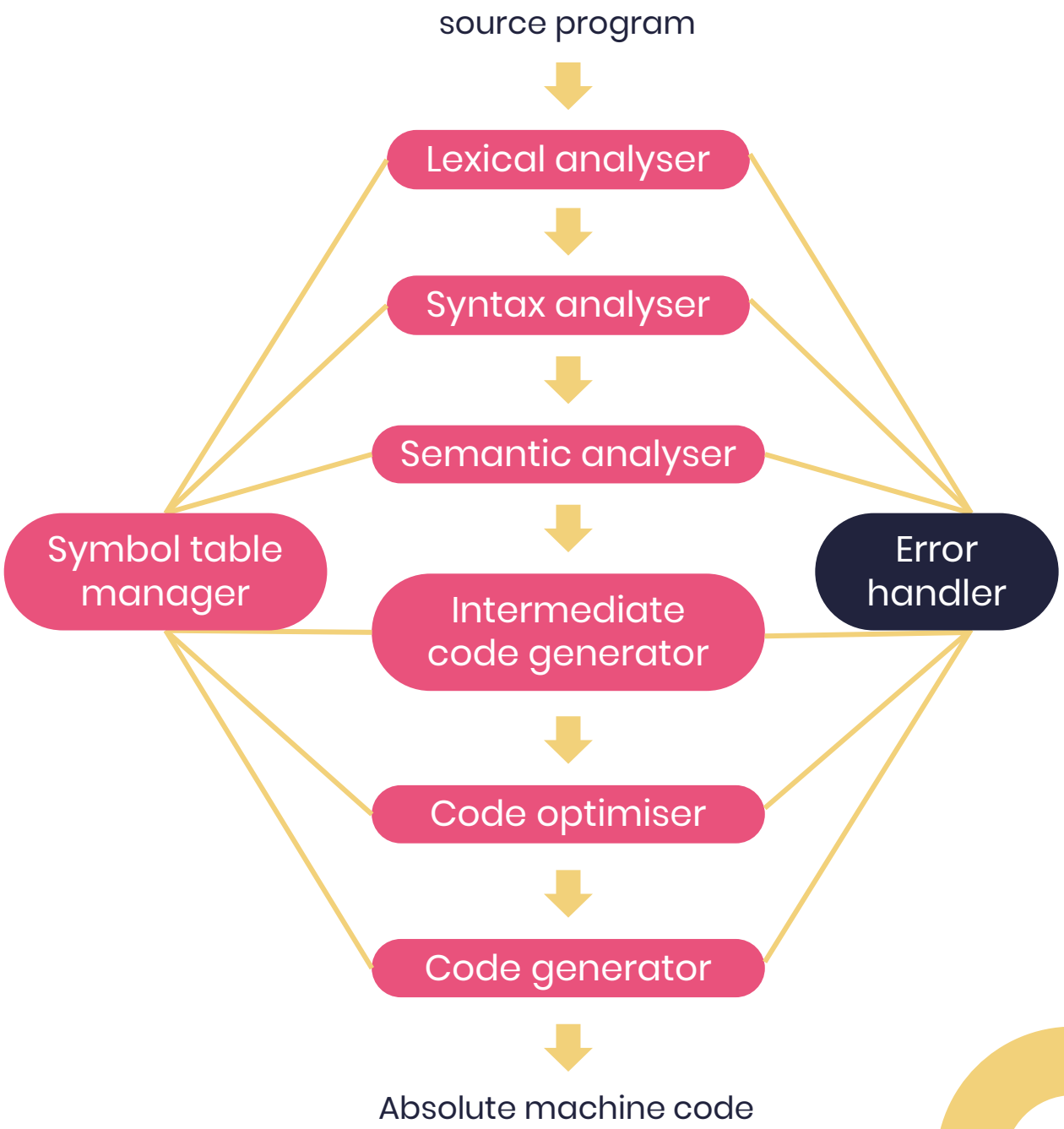
Semantic analysis

- Looks at code to see if it makes sense
- Checks variables



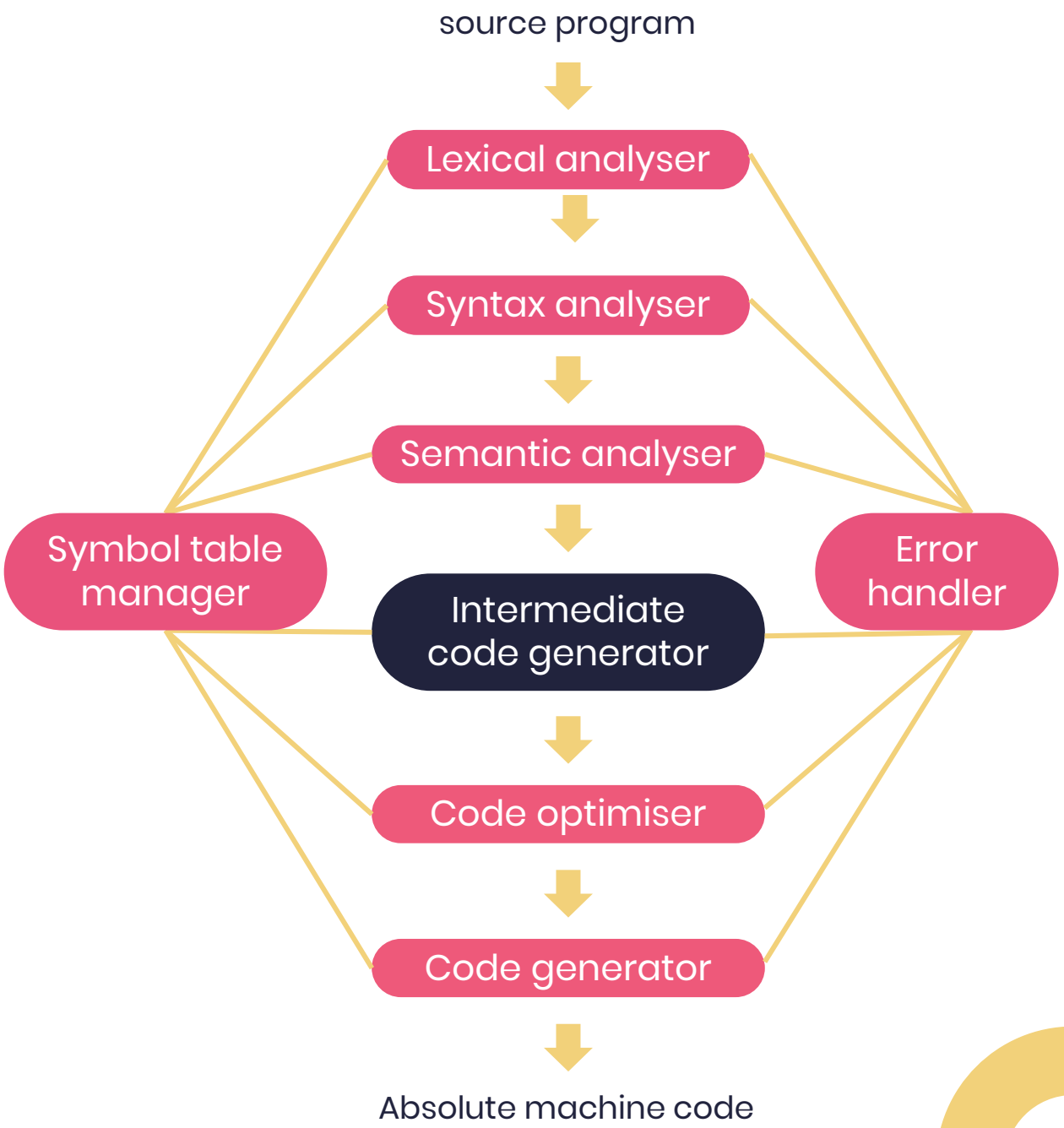
Symbol table

- Stores information about items in code
- So compiler knows when variables are unknown



Error handler

- All phases of compiler linked
- Reports all errors encountered during the compilation process



Intermediate code

- Code resembles assembly code
- Optimised in the code optimisation phase
- Generating smaller and faster machine code
- Translated into target machine code

Assembler

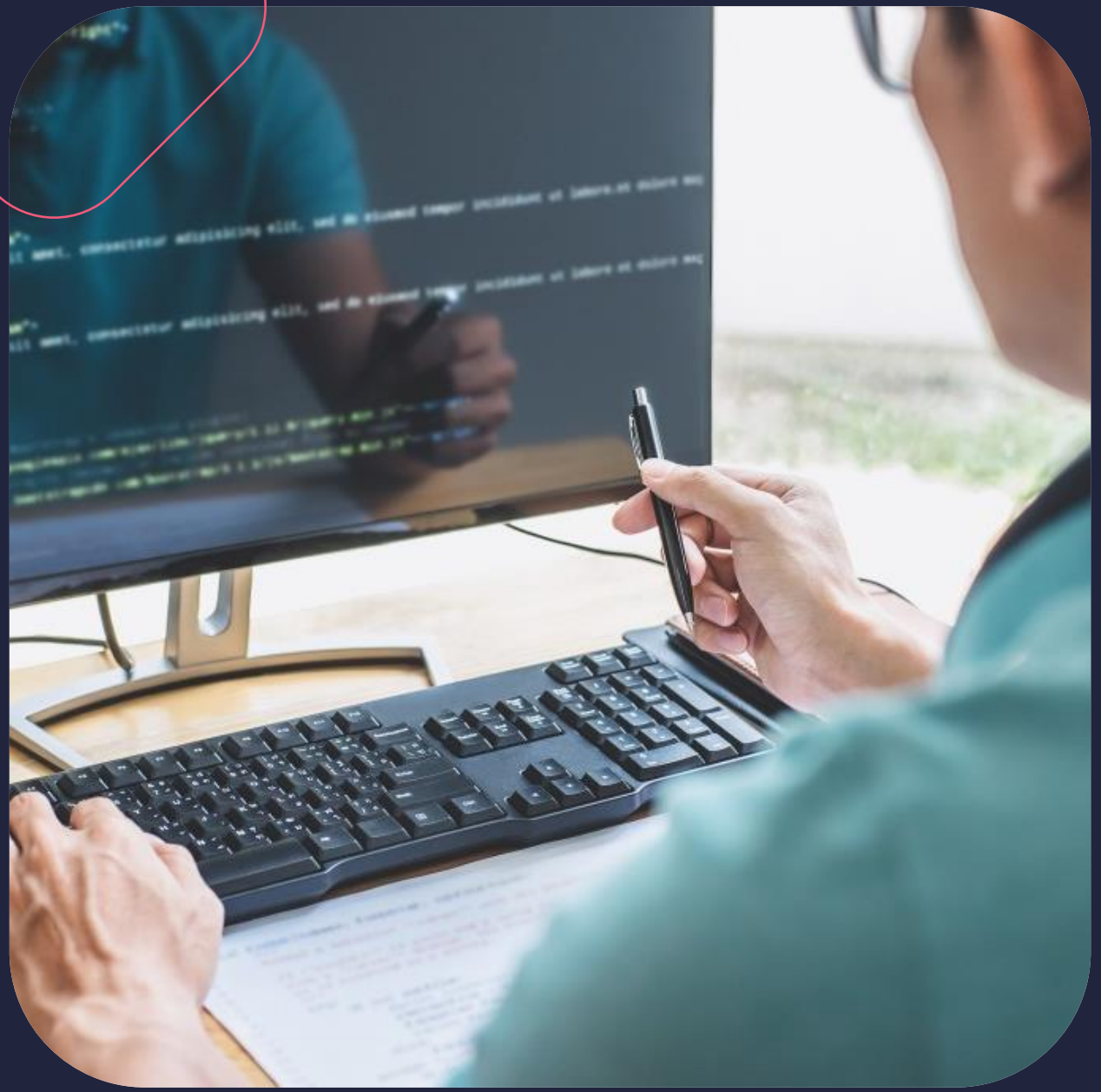
- Further translates the code to produce relocatable machine code
- First pass – the assembler reading the source file code once, takes note of the memory locations required and stores this information in a symbol table
- Second pass - the assembler reading the source file one more time and translating each operation code into the equivalent binary machine code
- Output is relocatable machine code

Loader/linker

- Relocatable machine code needs to be bundled together to produce one single executable file
- Linker collates all the bits of object code files
- Loader takes single object module as input and loads it into the system
- End result is an executable of the actual 1s and 0s loaded into memory and executed by the host computer

Absolute machine code

- File no longer portable - compiled for that machine type
- Write code for each system you intend to run it on
- Machine's instruction set and other factors affect the size of the resultant machine code



Recap

- Various types of programming languages
- Programming paradigms
- Two major levels of programming languages