Computer Languages

It’s all about the code!

Computer Languages

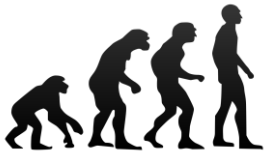
- Computers find it very difficult to understand natural (human) languages
- Computers are thus programmed using artificial languages designed for that specific purpose
- Some programming languages are general purpose:
  - COBOL
  - C
  - Java
- There are also many specialist programming languages
  - For Web programming (HTML5, XML, JavaScript, PHP)
  - For Database programming (SQL)
  - For Games programming (C++, C#)
  - For Data Analytics programming (R, Python)

Computer Languages

- There are several ways of categorising programming languages:
  - Assembly languages (Hardware & Processor-specific)
  - Functional languages (e.g. LISP)
  - Procedural languages (e.g. C, COBOL)
  - Object-Oriented languages (e.g. C++, Java)
  - Declarative languages (SQL)

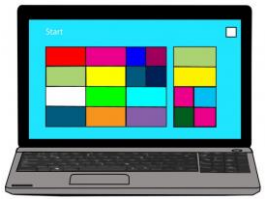
Language-Levels

- Languages also have different levels:



High-Level Languages

Mid-Level (Assembly Language)



Low-Level (Machine Code)

Human-oriented

Machine-oriented

Low-Level Language

Machine Code

- Written directly in binary (0/1s) format
- Exactly what the computer understands
- Can be executed immediately – so very fast
- Very difficult for people to write & understand
- Called '*programming to the metal*'
- The first generation of programming languages
- Used in industrial or low-level hardware settings
- Format directly tied to architecture of hardware

Mid-Level Language

Assembly Languages

- Written in ‘mnemonics’ (ADD, LOAD, HALT etc.)
- Codes represent machine code instructions
- *See example code on next slide…*
- (Relatively) easier than machine code for people to understand – BUT needs translating into machine code to run (covered later)
- Although higher level than machine code, it is still really low-level as format is tied to the hardware

Mid-Level Language

Assembly Languages

Machine code	Assembly code	Description
001 1 000010	LOAD #2	Load the value 2 into the Accumulator
010 0 001101	STORE 13	Store the value of the Accumulator in memory location 13
001 1 000101	LOAD #5	Load the value 5 into the Accumulator
010 0 001110	STORE 14	Store the value of the Accumulator in memory location 14
001 0 001101	LOAD 13	Load the value of memory location 13 into the Accumulator
011 0 001110	ADD 14	Add the value of memory location 14 to the Accumulator
010 0 001111	STORE 15	Store the value of the Accumulator in memory location 15
111 0 000000	HALT	Stop execution

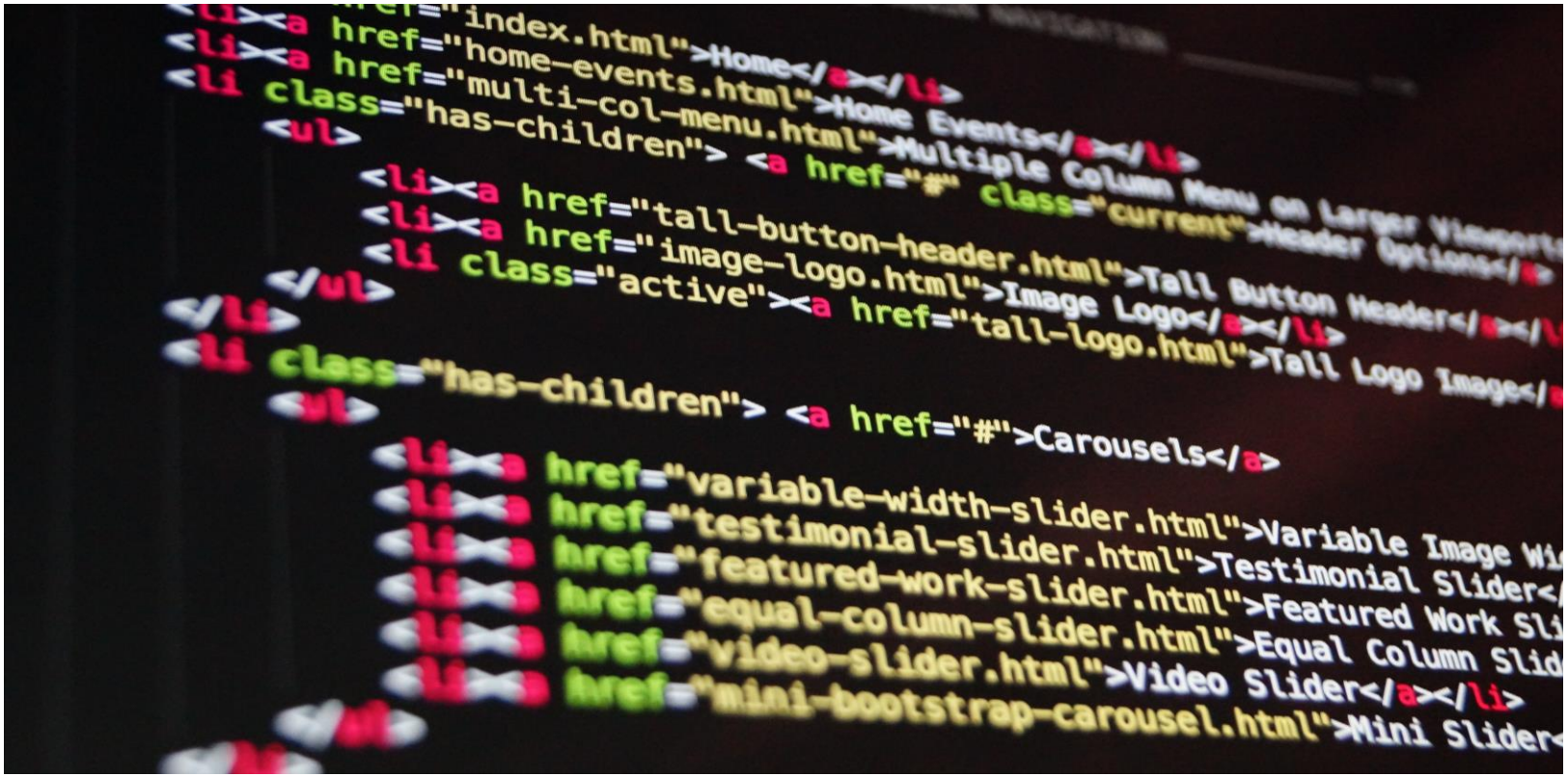
High-Level

High-Level languages

- Written in ‘English-like’ words and symbols
- *Much easier* for people to write and understand
- The vast majority of code is now written in high-level languages like Java, SQL, PHP, C++ etc.
- Needs significant translation to run (covered later)
- *See example code on next slide…*

High-Level

High-Level languages

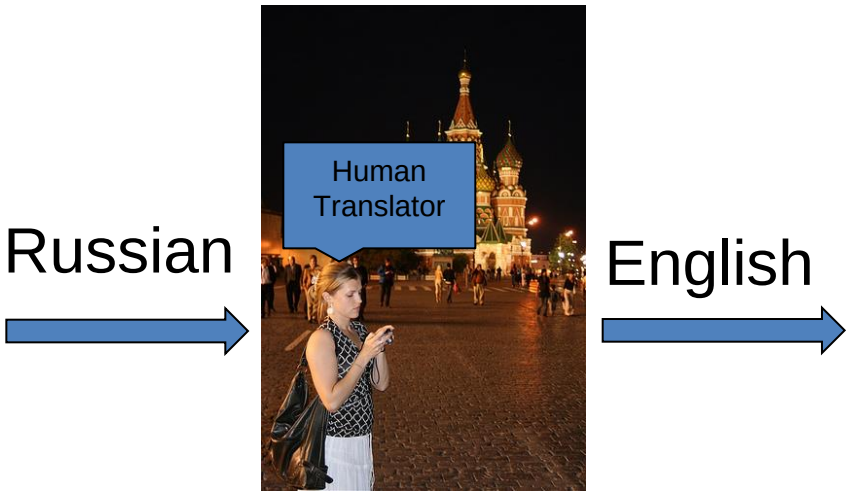


Language Translation

- Human programmers develop in *high-level* code
  - Java, C++, COBOL, C# etc.
- Computers do not understand this code
- Computers understand *low-level* machine code
  - 010101010 etc.
- We thus need a **translation process**
- Two approaches:
  - Interpreters
  - Compilers

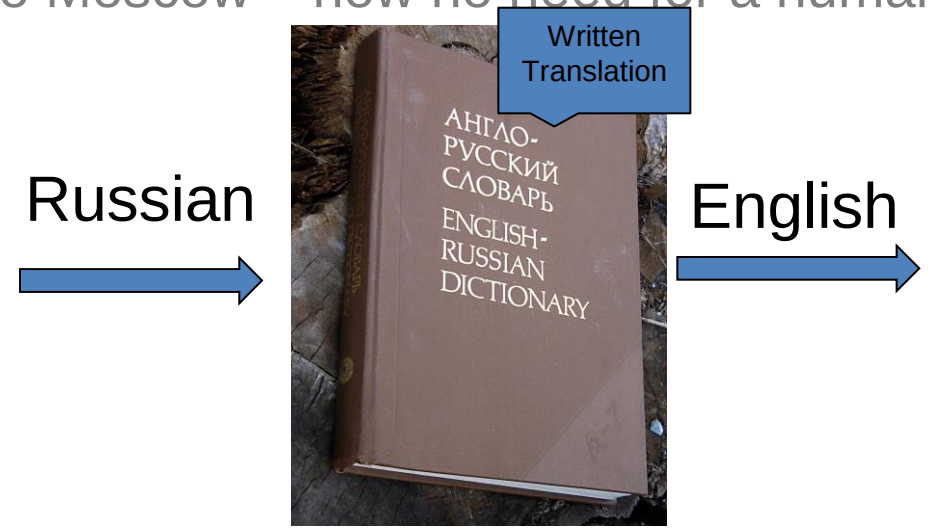
Interpreters

- **Real-time** conversion from high-level code to low-level code
- High-level (source) code **dynamically** converted to low-level (target) code – with *no intermediate saved file* – all done 'on-the-fly'
- Just like a human translator converting (say) Russian to English for an English tourist in Moscow

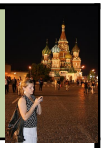


Compilers

- **Ahead-of-time** conversion from high-level to low-level
- High-level (source) code **statically** converted to low-level (target) code – using an *intermediate saved file*
- Just like the English tourist buying a Russian-to-English dictionary *before* flying to Moscow – now no need for a human translator



Translation Comparison

Compiler 	Interpreter 
Whole source code file converted at a time – ONCE.	Processes a single statement at a time - SLOWER
Save resultant executable file for later execution – READY TO GO	No executable file created or saved – REPEAT EACH TIME
Once translated, executable can be run multiple times - EFFICIENT	Need to translate every time it is run - INEFFICIENT
Easy to distribute converted executable file – EFFICIENT	No executable file to distribute could be SAFER?
Slow process during development – BUT ONLY DONE ONCE	Fast development time – BUT MUST BE REPEATED

Assemblers

- Same principle as compilers & interpreters – to **translate** a higher-level language down into machine code – so the computer can execute it
- An ‘assembler’ is a specialised computer program that translates assembly language code down into machine code

Machine code	Assembly code	Description
001 1 000010	LOAD #2	Load the value 2 into the Accumulator
010 0 001101	STORE 13	Store the value of the Accumulator in memory location 13
001 1 000101	LOAD #5	Load the value 5 into the Accumulator
010 0 001110	STORE 14	Store the value of the Accumulator in memory location 14
001 0 001101	LOAD 13	Load the value of memory location 13 into the Accumulator
011 0 001110	ADD 14	Add the value of memory location 14 to the Accumulator
010 0 001111	STORE 15	Store the value of the Accumulator in memory location 15
111 0 000000	HALT	Stop execution

Semantics V Syntax

- Two things can go wrong when writing software:
  - Your **logic** (algorithm design) is flawed
    - You have misunderstood the logic needed to solve the problem
    - These are known as **semantic** (meaning 'meaning') errors
    - You have tried to do something logically impossible such as:
      - » X/0 (Dividing by zero produces infinity!)
      - » IF (X > 100) AND (X < 100) THEN… (This is logically impossible)
  - Your **code** (program keywords) is flawed
    - The underlying logic is valid but the actual **syntax** is wrong…
      - » IFF (X > 50) THEN… (Should be IF)
      - » WHIL (X > 50) DO… (Should be WHILE)

Semantics V Syntax

- **Semantic** (logical) errors are hard for a computer to spot and may not be picked up in the translation process – relies on human intervention.
- Semantic errors often only noticed at **run-time**
- **Syntax** (spelling/format) errors are easy for a computer to spot and will be flagged-up during the translation process
- Syntax errors will never get to run-time – as they are detected during **translation process** and fixed

References

- https://opensource.com/resources/what-open-source
- https://opensource.org/
- https://www.gnu.org/software/software.en.html

Awarding Great British Qualifications

Topic 6 – Software, Installation and Configuration

Any Questions?