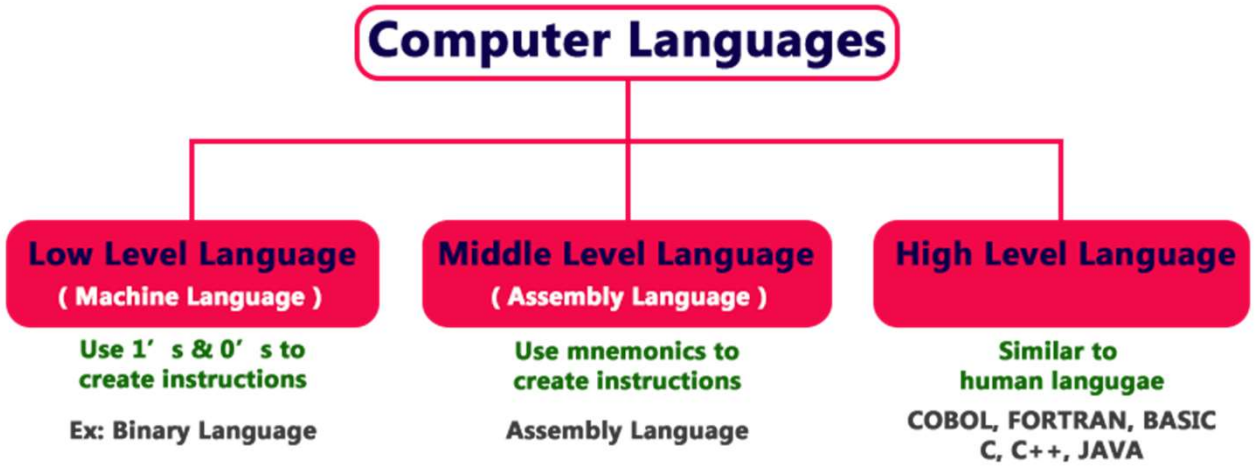
History of Programming

K.N.Toosi

Fall 2023

Categories of Programming Languages

- **Machine languages,** that are interpreted directly in hardware
- **Assembly languages,** that are thin wrappers over a corresponding machine language
- **High-level languages,** that are anything machine-independent



Machine Languages

- Machine language is the direct representation of the code and data run directly by a computing device. Machine languages feature:
  - Registers to store values and intermediate results
  - Very low-level machine instructions (add, sub, div, sqrt) which operate on these registers and/or memory
  - Labels and conditional jumps to express control flow
- The machine instructions are carried out in the hardware of the machine, so machine code is by definition **machine-dependent**. Different machines have different instruction sets. The instructions and their operands are all just bits.
- Machine code is usually written in hex. Here's an example for the Intel 64 architecture:

```
89 F8 A9 01 00 00 00 75 06 6B C0 03 FF C0 C3 C1 E0 02 83 E8 03 C3
```

Machine languages

- **Advantages**
  - A computer can easily understand the low-level language.
  - Low-level language instructions are executed directly without any translation.
  - Low-level language instructions require very less time for their execution.
- **Disadvantages**
  - Low-level language instructions are very difficult to use and understand.
  - Low-level language instructions are machine-dependent, that means a program written for a particular machine does not execute on another machine.
  - In low-level language, there is more chance for errors and it is very difficult to find errors, debug and modify.

Assembly Languages

- An assembly language is an encoding of machine code into something more readable. It assigns human-readable labels (or names) to storage locations, jump targets, and subroutine starting addresses, but doesn't really go too far beyond that.

```
        .globl  f
        .text
f:
        mov     %edi, %eax      # Put first parameter into eax register
        test    $1, %eax        # Examine least significant bit
        jnz     odd             # If it's not a zero, jump to odd
        imul    $3, %eax        # It's even, so multiply it by 3
        inc     %eax            # and add 1
        ret                     # and return it
odd:
        shl    $2, %eax         # It's odd, so multiply by 4
        sub    $3, %eax         # and subtract 3
        ret                     # and return it
```

Assembly Languages

- In assembly language, we use predefined words called **mnemonics**. Binary code instructions in low-level language are replaced with mnemonics and operands in middle-level language. But the computer cannot understand mnemonics, so we use a translator called **Assembler** to translate mnemonics into binary language. Assembler is a translator which takes assembly code as input and produces machine code as output.
- **Advantages**
  - Writing instructions in a middle-level language is easier than writing instructions in a low-level language.
  - Middle-level language is more readable compared to low-level language.
  - Easy to understand, find errors and modify.
- **Disadvantages**
  - Middle-level language is specific to a particular machine architecture, that means it is machine-dependent.
  - Middle-level language needs to be translated into low-level language.
  - Middle-level language executes slower compared to low-level language.

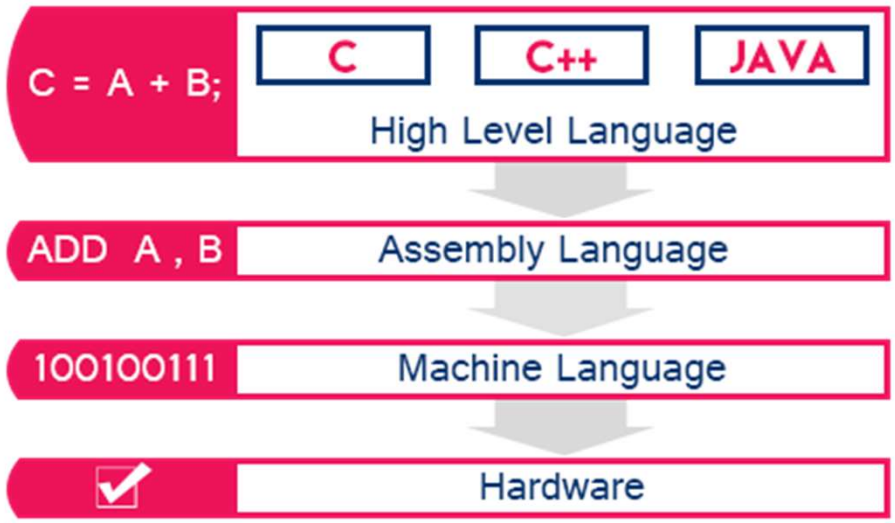
High-Level Languages

- High-level language is a computer language which can be **understood by the users**.
- The high-level language is very similar to human languages and has a set of **grammar rules** that are used to make instructions more easily.
- Every high-level language has a set of predefined words known as **Keywords** and a set of rules known as **Syntax** to create instructions.
- The high-level language is easier to understand for the users but the computer can not understand it. High-level language needs to be converted into the low-level language to make it understandable by the computer. We use **Compiler or inter**preter to convert high-level language to low-level language.

- Languages like FORTRAN,C, C++, JAVA, Python, etc., are examples of high-level languages. All these programming languages use human-understandable language like English to write program instructions.

High-Level Languages

- **Advantages**
  - Writing instructions in a high-level language is easier.
  - A high-level language is more readable and understandable.
  - The programs created using high-level language runs on different machines with little change or no change.
  - Easy to understand, create programs, find errors and modify.
- **Disadvantages**
  - High-level language needs to be translated into low-level language.

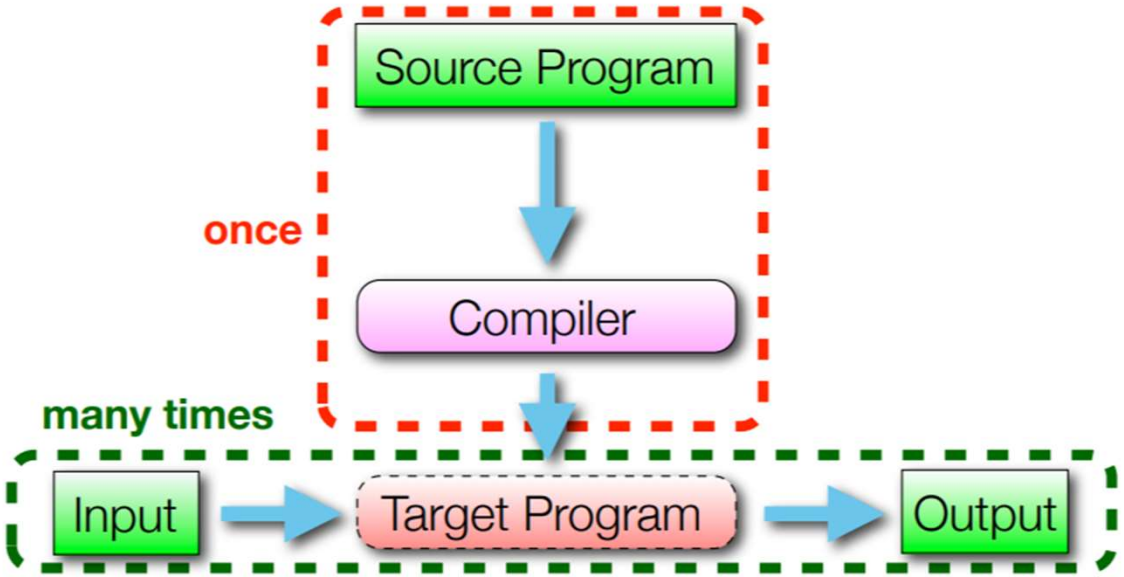

C = A + B;
C
C++
JAVA
High Level Language
ADD A , B
Assembly Language
100100111
Machine Language
Hardware

Executing High level languages

- High-level languages must be translated
- for execution:
- ➞ Ahead of execution: compilation.
- ➞ Piece-wise during execution: interpretation.

Compiler

- Ahead of time translation.
- ➞ From (high-level) source language to (lower-level) target language.
- ➞ Deep inspection of source program as a whole.
- ➞ Compiler is unaware of subsequent input.
- Translation occurs only **once**, but the program is executed **many times**.


Source Program
once
Compiler
many times
Input
Target Program
Output

Compliers

Advantages

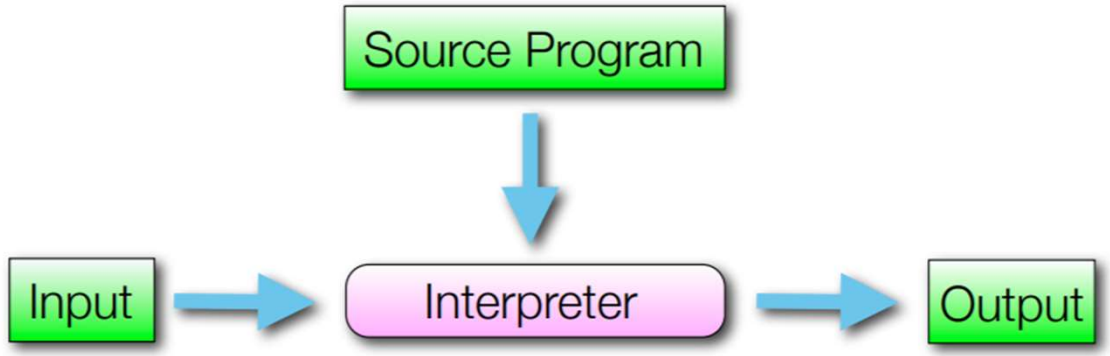
- No translation cost at runtime: efficient execution.
- Translation cost amortized over many runs.
- Can distribute program without revealing either source or compiler (commercial software distribution).

Disadvantages

- Runtimes error harder to diagnose.
- Slow edit-compile-test cycle (large systems can take minutes or hours to compile).
- Source may get lost (decompilation/reverse engineering is difficult and lossy).
- Good compilers are difficult to built.

Interpreters

- Translation during execution.
- ➞ **Each** run requires on-the-fly translation.
- ➞ Interpreter operates on two inputs: program and actual input.
- ➞ Often line/function/instruction interpreted individually on demand.



Interpreter

Advantages

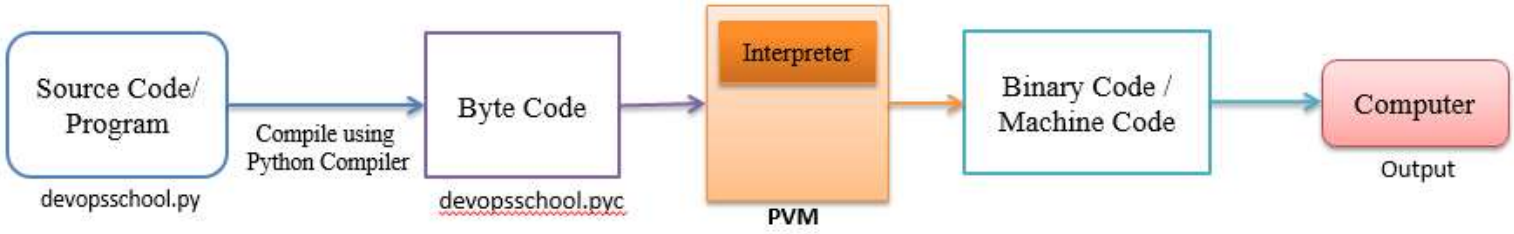
- Excellent debugging facilities: source code known when error occurs.
- Excellent checking: both input and source are known.
- Easy to implement.
- Can generate and evaluate new code at runtime (eval).

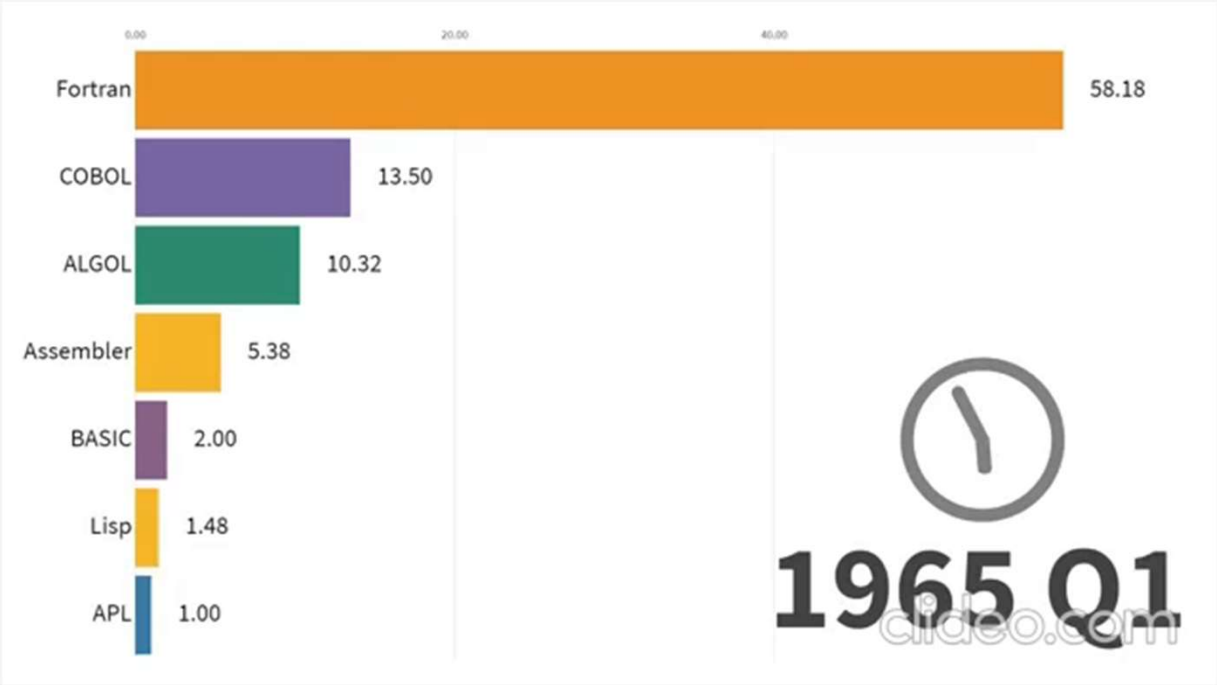
Disadvantages

- Translation occurs many times (redundant work).
- Translation cost occur at runtime: inefficient.
- Protecting intellectual property requires source code obfuscation (which can be unreliable).
- Reasonably fast interpreters are hard to implement.

So Python?

- interpreted/compiled is not a property of the language but a property of the implementation.
- Implementations of Python: **CPython**, Pypy, Jython
- CPython: First compile high-level to low-level byte code. Interpret much simpler byte code. Appears as interpreter to user.
- Enables "compile once, run everywhere".





From: https://www.youtube.com/watch?v=Og847HVwRSI

Converting decimal to binary

Algorithm for converting an ***integer*** decimal number:

1. Take decimal number as dividend.
2. Divide this number by 2.
3. Store the remainder in an array (it will be either 0 or 1).
4. Repeat the above two steps until the number is greater than zero.
5. Print the array in reverse order

   (which will be equivalent binary number of given decimal number).

Converting decimal to binary

Example – Convert decimal number 112 into binary number.

Division	Remainder (R)
112 / 2 = 56	0
56 / 2 = 28	0
28 / 2 = 14	0
14 / 2 = 7	0
7 / 2 = 3	1
3 / 2 = 1	1
1 / 2 = 0	1

Converting decimal to binary

- Procedure for converting an **fractional** decimal number:

1. Take decimal number as multiplicand.
2. Multiple this number by 2 (2 is base of binary so multiplier here).
3. Store the value of integer part of result in an array (it will be either 0 or 1).
4. Repeat the above two steps until the number became zero.
5. Print the array

   (which will be equivalent fractional binary number of given decimal fractional number).

Converting decimal to binary

Example – Convert decimal fractional number 0.8125 into binary number.

Multiplication	Resultant integer part (R)
0.81252 x 2= 1.625	1
0.625 x 2= 1.25	1
0.25 x 2= 0.50	0
0.50 x 2= 1.0	1
0 x 2 = 0	0

Now, write these resultant integer part, this will be 0.11010 which is equivalent binary fractional number of decimal fractional 0.8125.