Programming Languages: Classification, Execution Model, and Errors

4th Lecture

1. Introduction

As the involvement of computer, automation and robotics growing in our daily life, *programming becomes highly required to control all of them*. To control all of these systems and machines and take desired output by them ***skilled programming languages is necessary.*** However, the area of programming language becomes how much wide but it will be under one of the two categories of programming languages (i.e., Low-level language and High-level language). In the early days of computing, language design was heavily influenced by the decision to use compiling or interpreting as a mode of execution. *Depending on tools such as compilation and interpretation in order to get our written code into a form that the computer can execute.* Code can either be executed natively through the operating system after it is converted to machine code (via compilation) or can be evaluated line by line through another program which handles executing the code instead of the operating system itself (via interpretation).

2. Classification of Programming Languages

Programming languages are basically classified into two main categories – Low-level language and High-level language. Every programming language belongs to one of these categories and sub-category.

2.1. Low level languages

Low-level languages are used to write programs that relate to the specific architecture and hardware of a particular type of computer. They are closer to the native language of a computer (binary), ***making them harder for programmers to understand***. Programs written in low-level languages are **fast** and **memory efficient**. However, *it is nightmare* *for programmers to write, debug and maintain low-level programs.* They are mostly used to develop operating systems, device drivers, databases and applications that require direct hardware access. Low level languages are further classified in two more categories – Machine language and Assembly language.

- ❖ **Machine language**: *Machine language is closest language to the hardware. It consists set of instructions that are executed directly by the computer.* These instructions are a sequence of **binary bits**. Each instruction performs a very specific and small task. *Instructions written in machine language are machine dependent and varies from computer to computer*.
- ❖ **Assembly language**: *Assembly language is an improvement over machine language*. Similar to machine language, assembly language also interacts directly with the hardware. Instead of using a raw binary sequence to represent an instruction set, assembly **language uses mnemonics**. Assembly language uses a *special program called* **assembler**. The ***assembler*** translates mnemonics to specific machine code.

Advantages of low-level languages

- ✓ Programs developed using low-level languages are fast and memory efficient.
- ✓ Programmers can utilize processor and memory in a better way using a low-level language.
- ✓ There is no need of any compiler or interpreters to translate the source to machine code. Thus, cuts the compilation and interpretation time.
- ✓ Low-level languages provide direct manipulation of computer registers and storage.
- ✓ It can directly communicate with hardware devices.

Disadvantages of low-level languages

- Programs developed using low-level languages are machine dependent and are not portable.
- It is difficult to develop, debug and maintain.
- Low-level programs are more error-prone.
- Low-level programming usually results in poor programming productivity.
- A programmer must have additional knowledge of the computer architecture of a particular machine, for programming in the low-level language.

2.2. High level languages

High-level languages are *similar to the human language*. **high-level** *languages are programmers friendly, easy to code, debug and maintain. it provides a higher level of abstraction from machine language*. They do not interact directly with the hardware. Rather, they focus more on the complex arithmetic operations, optimal program efficiency and easiness in coding. Programs in a high-level language are written using English statements (such as Python, Java, C++, etc). High-level programs ***require compilers/interpreters*** to translate source code to machine language. We can compile the source code written in the high-level language to multiple machine languages. Thus, *they are machine independent language*. High-level languages are grouped into two categories based on the execution model – compiled or interpreted languages.

We can also classify high-level language several other categories based on the programming paradigm.

Structured programming (sometimes known as modular programming) *is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program* by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines. Hence, making it more efficient and easier to understand and modify. Structured programming frequently employs a **top-down design model**, in which developers map out the overall program structure into separate subsections. Note, it is possible to do structured programming in any programming language.

Procedural programming *is a programming paradigm, derived from structured programming, based upon the concept of the procedure call*. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by ***other procedures*** or ***itself***.

Object-oriented programming *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and* *code, in the form of procedures, often known as methods*. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated. Thus, programmers define not only the data type of a data structure but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, *programmers can create relationships between one object and another*.

Advantages of High-level language

- ✓ High-level languages are programmer friendly. They are easy to write, debug and maintain.
- ✓ It provide higher level of abstraction from machine languages.
- ✓ It is machine independent language.
- ✓ Easy to learn.
- ✓ Less error-prone, easy to find and debug errors.
- ✓ High-level programming results in better programming productivity.

Disadvantages of High-level language

- It takes additional translation times to translate the source to machine code.
- High-level programs are comparatively slower than low-level programs.
- Compared to low-level programs, they are generally less memory efficient.
- Cannot communicate directly with the hardware.

2.3. Differences between low level and high level programming language

	Low level language	High level language
1	They are faster than high level language.	They are comparatively slower.
2	Low level languages are memory efficient.	High level languages are not memory efficient.
3	Low level languages are difficult to learn.	High level languages are easy to learn.
4	Programming in low level requires additional knowledge of the computer architecture.	Programming in high level do not require any additional knowledge of the computer architecture.
5	They are machine dependent and are not portable.	They are machine independent and portable.
6	They provide less or no abstraction from the hardware.	They provide high abstraction from the hardware.

7	They are more error prone.	They are less error prone.
8	Debugging and maintenance is difficult.	Debugging and maintenance is comparatively easier.
9	They are generally used for developing system software's (Operating systems) and embedded applications.	They are used to develop a variety of applications such as – desktop applications, websites, mobile software's etc.

3. Interpreted and Compiled Languages

During the write of a program, you might need to decide whether to use a **compiled** language or an **interpreted** language for the ***program source code***. Both types of languages have their strengths and weaknesses. Usually, the decision to use an interpreted language is based on time restrictions on development or for ease of future changes to the program. *The terms interpreted language and compiled language are not well defined* because, in theory, any programming language can be either interpreted or compiled. In modern programming language implementation, it is increasingly popular for a platform to provide both options.

3.1. Interpreted language

An **interpreted** language *is a type of programming language for which most of its implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions*. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines, and then into another language (often machine code). Examples of some common interpreted languages include Ruby, JavaScript, and Python.

Interpreters take as input the abstract representation of the input program (in the source language), and evaluate it with relation to additional input data. The output of the interpreter is the output produced by the program being executed.

Advantages of Interpreted language

- easy to learn and use
- minimum programming knowledge or experience
- allows complex tasks to be performed in relatively few steps

- allows simple creation and editing in a variety of text editors
- allows the addition of dynamic and interactive activities to web pages
- edit and running of code is fast.

Disadvantages of Interpreted language

✓ usually run quite slowly
✓ limited access to low level and speed optimization code.
✓ limited commands to run detailed operations on graphics.

3.2. Compiled language

A **compiler** *is a program that converts human-readable code into computer-readable instructions—a process that only happens once in the lifespan of that code*. Initially, ***it takes a bit longer*** because the compiler has to rearrange, optimize, or "compile" object code first. Examples of purely compiled languages include C, C++ and Go.

A compiler transforms and optimizes a program written at a higher abstraction level (the source language) into a program written in a lower level of abstraction language (the target language). The target program is written in a *way that it can be executed by a machine, and whose instructions are of a finer grain and adapted to its internal data structures*.

Advantages of Compiled language

- fast execution
- optimised for the target hardware

Disadvantages of Compiled language

✓ require a compiler
✓ editing and deploying the code is a lot slower than interpreters.

4. Programming – Errors

Errors *are the mistakes or faults in the program that causes our program to behave unexpectedly and it is no doubt that the well versed and experienced programmers also makes mistakes.* Programming error ***are generally known as Bugs*** and the process *to remove bugs from program is called as* Debug/Debugging. There are basically three types of error: Compilation error or Syntax error, Runtime error or exception and Logical error.

4.1. Compilation error

Compilation errors *are the most common error occurred due to typing mistakes or if you don't follow the proper syntax of the specific programming language*. These error are thrown by the compilers and will prevent your program from running. These errors are most *common to beginners*. It is also called as Compile time error or Syntax error. These errors are easy to debug.

Example: Typing int as Int

4.2. Runtime error

Run Time errors (exception) *are generated when the program is running and leads to the abnormal behavior or termination of the program.* The general cause of Run time errors is *because your program is trying to perform an operation that is impossible to carry out*.

Example: Dividing any number by zero, Accessing any file that doesn't exist etc are common examples of such error.

4.3. Logical error

Logical errors *will cause your program to perform undesired operations which you didn't intended your program to perform*. These errors occur generally *due to improper logic used in program*. These types of errors are difficult to debug.

Example: Multiplying an uninitialized integer value with some other value will result in undesired output.

<Best Regards>

Dr. Raaid Alubady