RGPVNOTES.IN

Subject Name: **Principles of Programming Languages**

Subject Code: **IT-5002**

Semester: **5th**

Unit 1:

Language Evaluation Criteria

- **Readability** is the ease of which programs can be read and understood and can be directly related to:
  - The overall simplicity, which can be determined by number of basic components and whether it has one or more ways to accomplish a particular operation
  - The orthogonality, which is the degree to which a set of primitive constructs can be combined to build the control and data structures of the language
  - The number of control statements in a language
  - The number of data types and structures
  - The syntax or form of the elements of a language. Three types of syntax affect readability: identifier forms, special words, and form and meaning
- **Writability** is a measure of how easily a language can be used to create programs for a chosen problem domain and can be directly related to:
  - Simplicity and orthogonality (defined in readability)
  - Support for abstraction, which is the ability to define and use complicated structures or operations in a way that allows many details to be ignored
  - Expressivity, which is when a language has a convenient way of specifying computations
- **Reliability** is when a language performs to its specifications under all conditions and is directly related to:
  - Type checking, which is simply testing for type errors in a program either by the compiler or run-time
  - Exception handling, which is the ability of a program to intercept run-time errors, take corrective action, and then continue on
  - Aliasing, which is having two or more distinct referencing methods or names for the same memory location
  - Readability and Writability

Influences on Language Design

1. Computer Architecture: By 1950, the basic architecture of digital computers had been established (and described nicely in John von Neumann's EDVAC report). A computer's machine language is a reflection of its architecture, with its assembly language adding a thin layer of abstraction for the purpose of making easier the task of programming. When FORTRAN was being designed in the mid to late 1950's, one of the prime goals was for the compiler to generate code that was as fast as the equivalent assembly code that a programmer would produce "by hand". To achieve this goal, the designers simply put a layer of abstraction on top of assembly language, so that the resulting language still closely reflected the structure and operation of the underlying machine.

The style of programming exemplified by FORTRAN is referred to as imperative, because a program is basically a bunch of commands. (Recall that, in English, a command is referred to as an "imperative" statement, as opposed to, say, a question, which is an "interrogative" statement.)

This style of programming has dominated for the last fifty years. Granted, many refinements have occurred. In particular, Object Oriented languages put much more emphasis on designing a program based upon the data involved and less on the commands/processing. But the notion of having variables (corresponding to memory locations) and changing their values via assignment commands is still prominent.

2. Programming Methodologies: Advances in methods of programming also have influenced language design, of course. Refinements in thinking about flow of control led to better language constructs for selection (i.e., if statements) and loops that force the programmer to be disciplined in the use of jumps/branching (by hiding them). This is called structured programming.

An increased emphasis on data (as compared to process) led to better language support for data abstraction. This continued to the point where now the notions of abstract data type and module have been fused into the concept of a class in object-oriented programming.

Language Categories

- Imperative – based on the von Neumann architecture
- Functional – based on mathematical functions
- Logic – rule based
- Object oriented – object based

Programming paradigms

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Imperative programming in which the program is built from one or more procedures (also termed subroutines or functions). The concept of imperative programming is

"First do this and next do that"

Characteristics of imperative programming:

- Discipline and idea
  - Digital hardware technology and the ideas of Von Neumann
- Incremental change of the program state as a function of time.
- Execution of computational steps in an order governed by control structures
  - The steps for commands
- Straightforward abstractions of the way a traditional Von Neumann computer works
- Similar to descriptions of everyday routines, such as food recipes and car repair
- Typical commands offered by imperative languages
  - Assignment, IO, procedure calls
- Language representatives
  - Fortran, Algol, Pascal, Basic, C
- The natural abstraction is the procedure
  - Abstracts one or more actions to a procedure, which can be called as a single command.
  - "Procedural programming"

The object-oriented paradigm is probably the conceptual anchoring of the paradigm. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. In that way, all the necessary technicalities of programming come in second row. The basic concept of OO is

"Send messages between objects to simulate the temporal evolution of a set of real world phenomena"

Characteristics of Object Oriented Programming:

- Discipline and idea
  - The theory of concepts, and models of human interaction with real world phenomena
- Data as well as operations are encapsulated in objects
- Information hiding is used to protect internal properties of an object
- Objects interact by means of message passing
  - A metaphor for applying an operation on an object
- In most object-oriented languages objects are grouped in classes
  - Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects
  - Classes represent concepts whereas objects represent phenomena
- Classes are organized in inheritance hierarchies
  - Provides for class extension or specialization

Functional programming is in many respects a simpler and more clean programming paradigm than the imperative one. The reason is that the paradigm originates from a purely mathematical discipline: the theory of functions. The imperative paradigm is rooted in the key technological ideas of the digital computer, which are more complicated, and less 'clean' than mathematical function theory. The basic concept of functional programming is

"Evaluate an expression and use the resulting value for something"

Characteristics of functional programming:

- Discipline and idea
  - Mathematics and the theory of functions
- The values produced are non-mutable
  - Impossible to change any constituent of a composite value
  - As a remedy, it is possible to make a revised copy of composite value
- A temporal
  - Time only plays a minor role compared to the imperative paradigm
- Applicative
  - All computations are done by applying (calling) functions
- The natural abstraction is the function
  - Abstracts a single expression to a function which can be evaluated as an expression
- Functions are first class values
  - Functions are full-fledged data just like numbers, lists, ...
- Fits well with computations driven by needs
  - Opens a new world of possibilities

The logic paradigm is dramatically different from the other three main programming paradigms. The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation. The basic concept is

"Answer a question via search for a solution"

Characteristics of the logic programming paradigm:

- Discipline and idea
  - Automatic proofs within artificial intelligence
- Based on axioms, inference rules, and queries.
- Program execution becomes a systematic search in a set of facts, making use of a set of inference rules

Programming Language Implementation

It is a system for executing computer programs. There are two general approaches to programming language implementation:

Interpretation: An interpreter takes as input a program in some language, and performs the actions written in that language on some machine.

Compilation: A compiler takes as input a program in some language, and translates that program into some other language, which may serve as input to another interpreter or another compiler.

A compiler does not directly execute the program. Ultimately, in order to execute a program via compilation, it must be translated into a form that can serve as input to an interpreter.

When a piece of computer hardware can interpret a programming language directly, that language is called machine code. A so-called native code compiler is one that compiles a program into machine code. Actual compilation is often separated into multiple passes, like code generation (often for assembler language), translator (generating native code), linking, loading and execution.

If a compiler of a given high level language produces another high level language, it is called **translator** (source to source translation), which is often useful to add extensions to existing languages or to exploit good and portable implementation of other language (for example C).
Many combinations of interpretation and compilation are possible, and many modern programming language implementations include elements of both. For example, the Smalltalk programming language is conventionally implemented by compilation into bytecode, which is then either interpreted or compiled by a virtual machine (most popular ways is to use JIT or AOT compiler compilation. This implementation strategy has been copied by many languages since Smalltalk pioneered it in the 1970s and 1980s.

Compilation and Virtual Machines
A virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. Their implementations may involve specialized hardware, software, or a combination.
There are different kinds of virtual machines, each with different functions:
System virtual machines (also termed full virtualization VMs) provide a substitute for a real machine. They provide functionality needed to execute entire operating systems. A hypervisor uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine. Modern hypervisors use hardware-assisted virtualization, virtualization-specific hardware, primarily from the host CPUs.
Process virtual machines are designed to execute computer programs in a platform-independent environment.
Some virtual machines, such as QEMU, are designed to also emulate different architectures and allow execution of software applications and operating systems written for another CPU or architecture. Operating-system-level virtualization allows the resources of a computer to be partitioned via the kernel's support for multiple isolated user space instances, which are usually called containers and may look and feel like real machines to the end users.
Instead of compiling the source code for the respective OS (on which it is targeted), you compile once and run everywhere.
For the sake of this question, I would call it VM (for example, both for Java and .NET). So this the execution of programs becomes something like
| Executable | -> | VM | -> | OS |
The compilers remain generic for the respective VM. However, the implementation of VM may vary depending on the machine it is going to be installed i.e. (*nix, windows, mac) x (32 bit, 64 bit).

We cannot use compiler instead of Virtual Machine because we no longer have a portable executable; we would have one executable for each platform. The user would have to either download a specific executable for their platform, or compile the code themselves on their specific platform.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in