1. Introduction

Q.3.2.1.1 Explain the notion of system software.
Answer: System software is a collection of programs that bridge the gap between the users wish to interact with the computer and the level at which the computer is capable of operating. These programs act as intermediary between users and the computer. Many a times, users' programs are translated by the system software into a form that computer can understand it, and programs get executed. Examples of system software are operating system, compiler, linker, loader, assembler, interpreter and database management system.

Q.3.2.1.2 Discuss different types of system software.
Answer: There are different types of system software as noted below.
∗ *Operating system* (OS): It is an interface between computer hardware and end user. Examples are Windows 10, Mac OS X, and Ubuntu.
∗ *Device driver*: Drivers make it possible for all connected components and external add-ons perform their intended tasks as directed by the OS. Examples of devices which require drivers are mouse, keyboard, soundcard, display card, network card and printer.
∗ *Firmware*: Firmware is a specific class of software that provides the low-level control for a device's specific hardware. Firmware can either provide a standardized operating environment for more complex device software, or, act as the device's complete operating system for less complex devices. Firmware is held in non-volatile memory devices such as ROM, EPROM, EEPROM and flash memory. Firmware such as BIOS of a personal computer may contain only basic functions of a device and may only provide services to higher-level software.
∗ *Language processors*: See Q.3.2.2.3.
∗ *Utilities*: Refer to Q.3.2.1.5.

Q.3.2.1.3 What do you mean by semantic gap? How does it affect to software development process?
Answer: It can be described as the gap between the task to be performed at application level and the execution level (machine level). It has two components: specification gap and execution gap.
Semantic gap = Specification gap + Execution gap
Specification gap refers to the gap between the task at application level

and the program written in some language. Execution gap refers to the gap between the program written in a language and the execution level (machine level) of a computer.

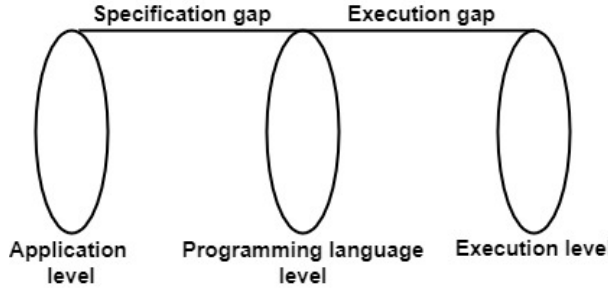

Figure 1.1: Specification gap and execution gap

The semantic gap may result in the following issues: (i) long development time, (ii) large development efforts for engineering a large-scale software.

Q.3.2.1.4 Fill in the blank.
A system software is a collection of programs, called ——.
Answer: system programs

Q.3.2.1.5 Classify software based on the nature of the task and goal. Draw a hierarchy chart.
Answer: Computer software includes various computer programs, system libraries and their associated documentation. Based on the nature of the task and goal, computer software can be classified broadly into application software, utility software, and system software. See Fig. 1.2. A description about *system software* is given in Q.3.2.1.1.
Utility software: This type of software is designed to help to analyze, configure, optimize or maintain a computer. Although a basic set of utility programs is usually distributed with an operating system (OS), and this first party utility software is often considered part of the operating system, users often install replacements or additional utilities. Those utilities may provide additional facilities to carry out tasks that are beyond the capabilities of the operating system. Example of utility programs are anti-virus software, firewalls, and disk utilities.
Application software: Application software (app for short) is a program or group of programs designed for end-users. Examples of an application include a word processor, a spreadsheet, an accounting application,

a web browser, an email client, a media player, a file viewer, simulators, a console game, or a photo editor.

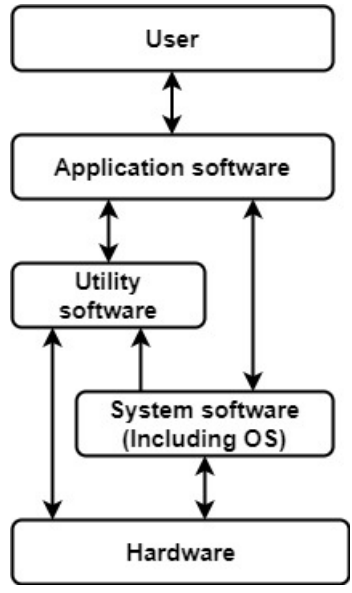

Figure 1.2: Software hierarchy based on the nature of the task and goal

Application software is concerned with the solution of some problems; it uses a computer as a tool and enables the end user to perform specific and productive tasks.

Q.3.2.1.6 What is system programming? Characterize the nature of system programming.
Answer: A system program is characterized by the fact that it is aimed at producing system software that provides services to the computer hardware, or specialized system services. Many a times, system programming directly deals with the peripheral devices with a focus on input, process (storage), and output.
Some characteristics of system programming include the following:
∗ Programmers are expected to know the hardware and internal behavior of the computer system on which the program will run. System programmers explore hardware properties and write software for specific hardware.
∗ It uses a low level programming language or some programming dialect.

2. Language Processors

Q.3.2.2.1 What do you mean by a language translator?
Answer: It is a software which bridges an execution gap. The input program of a language translator is a source program and the output program is the target program. The different types of language translator are mentioned below.
∘ *Assembler*: A language translator whose source language is an assembly language.
∘ *Compiler*: A language translator whose source language is a high level language.
∘ *Detranslator*: A language translator which converts machine language to an assembly level language.

Q.3.2.2.2 Write a note on interpreter.
Answer: An interpreter is a language processor which bridges an execution gap without generating a machine language program. It is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies to execute a program.
∗ Parse the source code and execute it directly.
∗ Translate source code into some efficient intermediate representation, and then immediately execute the intermediate representation.
∗ Explicitly execute stored precompiled code produced by a compiler.

Q.3.2.2.3 What is a language processor? Explain the purpose of the following language processors: preprocessor, language migratory.
Answer: It is a software which bridges a specification or execution gap. The input program of a language processor is a source program and the output program is the target program. Language translators (see Q.3.2.2.1) are also a kind of language processor. We define below two other types of language processors.
∘ *Preprocessor*: It is a software that accepts a source program containing directives and bridges the specification gap and produces a target program written in a language other than machine language.
∘ *Language migratory*: It is a software that bridges the specification gap between two programming languages.

Q.3.2.2.4 Give the specific names of the following softwares (language processors) that convert a
(i) COBOL program into a C program
(ii) C++ program into a C program
(iii) C program into a C program without directives (like # define)
(iv) 8088 assembly program into a machine language program.
(v) C++ program into machine language program.
Answer: (i) language migrator, (ii) preprocessor, (iii) preprocessor, (iv) assembler, (v) compiler.

Q.3.2.2.5 Make a comparison between problem oriented languages and procedure oriented languages.
Answer: A problem oriented language is designed to handle a particular class of problem. For example, COBOL was designed for business data processing, FORTRAN for scientific calculations and GPSS for simulation & modeling.
A procedure, often termed as a routine or subroutine, contains a series of computational steps to be carried out. A given procedure might be called at any point during a program's execution, including by other procedures or itself. A procedure oriented language follows structured programming, based on the concept of the procedure call. Example of procedure oriented language are FORTRAN, ALGOL, Pascal and C. See Table 2.1 for a comparison between the two types of languages.

Table 2.1 Problem oriented language versus procedure oriented language

Problem oriented language	*Procedure oriented language*
i. very close to application domain	i. independent of application domain
ii. small specification gap	ii. large specification gap
iii. large execution gap	iii. relatively small execution gap
iv. many times they are interpreted	iv. many times they are compiled

Q.3.2.2.6 Compare and contrast translation model with interpretation model.
Answer: We present a comparative study using Table 2.2.

Table 2.2 Comparison between translation model and interpretation model

Translation model	*Interpretation model*
i. translation overloaded before the program executes	i. no translation overhead the program executes
ii. program executes efficiently and faster	ii. program executes slower
iii. it is advantageous if a program does not changes frequently	iii. it is advantageous if a program is modified between executions

Q.3.2.2.7 Discuss, in general, the work performed by a language processor.
Answer: Language processing can be thought of as two-step process as shown in Fig. 2.1:
Language processing = Analysis of source program + Synthesis of target program.

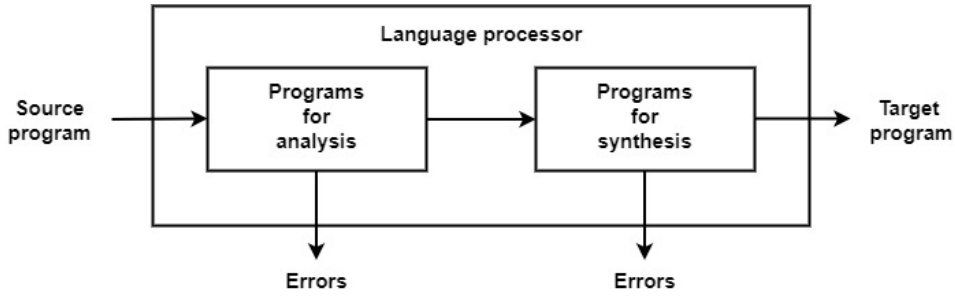

Figure 2.1: Major phases of a language processor

Now the analysis of source language consists of the following steps:
1. Lexical analysis: It is the process of converting a sequence of characters, as stated in a computer program or web page, into a sequence of tokens. A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner.
2. Syntax analysis: It is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.
3. Semantic analysis: It is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking, or makes sure a variable is declared before use which is impossible to describe in the extended Backus-Naur form and thus, not easily detected during parsing.

3. Assemblers

Q.3.2.3.1 Explain different entities in an assembly language program.
Answer: Assembly language program contains absolute entities, relative entities and external entities.
Absolute entity: It is independent of the storage location. E.g., op-code, fixed address (e.g., register).
Relative entity: It is relative to other symbolic references and can be stated relative to the starting address of the program. E.g., symbolic reference.
External entity: This type of entity is used in a module but not defined within that module.

Q.3.2.3.2 Is assembly language program portable?
Answer: When a source program written in a language and that can be compiled and run on a wide variety of computer systems is said to be portable. An assembly language program is not portable, because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family. Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370.

Q.3.2.3.3 State some situations, where assembly language programs become useful.
Answer: In the following situations, assembly language programs become useful.
∗ Assembly language is an ideal tool for writing embedded programs in single-purpose devices such as telephones, air-conditioning control systems, security systems, video cards, sound cards, hard drives, modems,and printers, because of its economical use of memory.
∗ Real-time applications dealing with simulation and hardware monitoring require precise timing and responses. Programs written in assembly language can achieve these objectives.
∗ Computer game consoles require their software to be highly optimized for small code size and fast execution. Software require to utilize the computer hardware to optimise speed and code size. Assembly language programs can take full advantage of these requirements.
∗ Device drivers contain a significant amount of assembly language code.

∗ For testing architectural features of a computer, assembly language programs become handy.

Q.3.2.3.4 State functions performed by an assembler.
Answer: Some important functions performed by assembler are given below:
(i) It produces the object code of the given program.
(ii) It prepares a cross reference table to assist in debugging.
Cross reference table indicates the following:
For each symbol in the source program, it displays where it is defined and where it is accessed.

Q.3.2.3.5 Explain the notion of system virtual machine.
Answer: A virtual machine has no direct correspondence to any real hardware. The physical, "real-world" hardware running the virtual machine is generally referred to as the "host", and the virtual machine emulated on that machine is generally referred to as the "guest". A host can emulate several guests, each of which can emulate different operating system / language platforms.

Q.3.2.3.6 Discuss different levels of virtual machine.
Answer: Different levels of virtualization of a machine are described below (see Fig.3.1).
Instruction set architecture (ISA) is an abstract model of a computer. It specifies basic operations, supported data types, registers, hardware support for managing main memory, fundamental features (such as the memory consistency, addressing modes, virtual memory), and input/output model of a family of implementations of the ISA. The set of instructions is also referred to as machine language.
Assembly language is kept above the ISA level. Assembly language uses short mnemonics such as ADD, SUB, and MOV, which are easily translated to the ISA level. Assembly language programs are translated (assembled) into their entirety into machine language before they begin to execute.
High level languages such as C, Java and Pascal are kept at the level 4. A high level program gets translated into assembly language program, and then it gets converted into a machine language program. Finally, the machine language program starts executing.

Q.3.2.3.7 What is an assembler directive?

Answer: An assembler directive instructs assembler to perform certain actions during process of assembling program. E.g., ORIGIN 2000 directs LC (Location Counter) to be set to 2000 address. Object program starts generating code from that address onwards. The ORIGIN statement is useful when target program does not contain consecutive memory words.

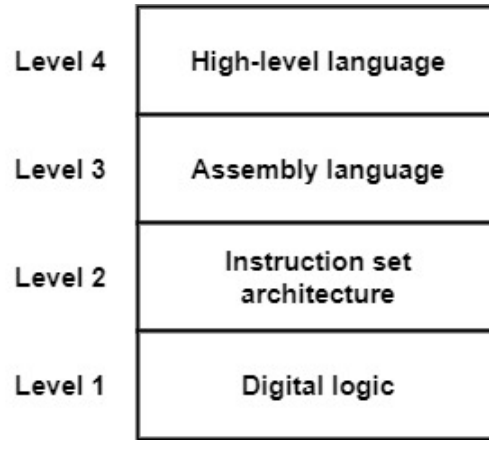

Figure 3.1: Virtual machine levels

Q.3.2.3.8 What is a literal? Explain how the following commands are processed by an assembler: (i) *ADD A*, (ii) *ADD* @21.
Answer: An operand, whose value is literally stated, is called a literal.
(i) Let *A* be a memory location. At the end of pass I, the assembler gets the address of *A*. In the pass II, the assembler replaces ADD by its corresponding machine code from machine op-code table. Then *A* is replaced by its address from symbol table (ST).
(ii) @21 refers that 21 is a literal. 21 has to be directly added with the accumulator. In the pass I, when the assembler gets the symbol 21 preceeded by @, then 21 is treated as a literal and put in the literal table. At this time, the assembler associates a memory location for the literal 21, and the assembler puts value of literal 21 as 21 in the symbol table. In the pass II, the assembler replaces *ADD* by its corresponding machine code from machine op-code table. Then literal 21 is replaced by its address from literal table.

Q.3.2.3.9 What is load-and-go assembler?
Answer: Load-and-go assembler accepts input (i.e., an assembly language program) and produces output in the main memory. It becomes