Unit 1

Introductory Concepts

Why Study Programming Languages?

- Any notation for the description of algorithms and data structures may be termed a programming language. There are six primary reasons for the study of programming languages.
- **Reason 1 – To improve your ability to develop effective algorithms.** For example, recursion supports direct implementation of elegant and efficient algorithms.
- **Reason 2 – To improve your use of your existing programming language.** By understanding how features in your language are implemented, you greatly increase your ability to write efficient programs.
- **Reason 3 – To increase your vocabulary of useful programming constructs.** By studying the constructs provided by a wide range of languages, a programmer increases his programming vocabulary.

Why Study Programming Languages?

- **Reason 4 – To allow a better choice of programming language.** A knowledge of a variety of languages may allow the choice of just the right language for a particular project, thereby reducing the required coding effort. For example, developing applications useful in decision making, such as in artificial intelligence applications, would be more easily written in LISP, Prolog, or Python.
- **Reason 5 – To make it easier to learn a new language.** A thorough knowledge of a variety of programming language constructs and implementation techniques allows the programmer to learn a new programming language more easily when the need arises.
- **Reason 6 – To make it easier to design a new language.** Language design is often simplified if the programmer is familiar with a variety of constructs and implementation methods from ordinary programming languages.

Short History of Programming Languages

- Programming language designs and implementation methods have evolved continuously since the earliest high-level languages appeared in the 1950s.
- The first versions of FORTAN and LISP were designed during the 1950s. Ada, C, Pascal, Prolog, and Smalltalk date from the 1970s. C++, ML, Perl, and Postscript date from the 1980s. Java dates from the 1990s.
- In the 1960s and 1970s, new languages were often developed as part of major software development projects.
- **Development of Early Languages:**
  - **Numerically based languages:**
    - Grace Hopper developed A-0 (Arithmetic Language version 0) language in 1951 for UNIVAC I. The A-0 system was followed by the A-1, A-2, A-3, AT-3 and B-0.

Short History of Programming Languages

- John Backus developed Speedcoding language for IBM 701 in 1953
- In 1957, FORTRAN was originally developed by John Backus at IBM for scientific calculations; FORTRAN was revised as FORTAN II in 1958 and FORTRAN IV a few years later; In 1966, FORTRAN IV became a standard under the name FORTRAN 66 and has been upgraded twice since to FORTRAN 77 and FORTRAN 90.
- ALGOL 58, originally named IAL (International Algorithmic Language) was originally developed in 1958. A revision occurred in 1960, and ALGOL 60 (with a minor revision in 1962) became the standard academic computing language from the 1960s to the early 1970s. ALGOL 68 developed in *1968 as* a successor to the ALGOL 60.
- Jules Schwartz developed JOVIL (*Jules' Own Version of the International Algorithmic Language)* to compose software for the electronics of military aircraft in 1959.

Short History of Programming Languages

- Simula (Simula I and Simula 67) was developed in the 1960s. Simula 67 introduced object-oriented concepts.
- Wirth developed ALGOL-W in the mid-1960s as an extension to ALGOL. Around 1970, he developed Pascal, which became the computer science language of the 1970s.
- C programming language was developed in 1972 by Dennis Ritchie at bell laboratories. C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories as an extension of C language with object-oriented features.
- IBM developed NPL (New Programming Language) in 1963 at its Hursley Laboratory in England. Its name was later changed to MPPL (Multi-Purpose Programming Language), which was then shortened to just PL/I. The educational subset PL/C achieved modest success in the 1970s.
- The original version of BASIC (Beginners' All-purpose Symbolic Instruction Code) language was created by John G. Kemeny and Thomas E. Kurtz at Dartmouth College in 1963.

Short History of Programming Languages

- **Business languages:**
  - Grace Hopper led a group at Univac to develop FLOWMATIC in 1955 to develop business applications.
  - In 1959, the U.S. Department of Defense sponsored a meeting to develop Common Business Language (BCL) and the specifications published in 1960 as COBOL (COmmon Business Oriented Language).
  - COBOL was revised in 1961 and 1962, standardized in 1968, and revised again in 1974 and 1984.
- **Artificial-intelligence languages:**
  - Information Processing Language (IPL) was created by Allen Newell, Cliff Shaw, and Herbert A. Simon at RAND Corporation and the Carnegie Institute of Technology about 1956.
  - Several versions of IPL were created: IPL-I, IPL-II, IPL-III, IPL-IV, IPL-V, and IPL-VI.

Short History of Programming Languages

- LISP (list processing) language was developed about 1960 by John McCarthy at the Massachusetts Institute of Technology (MIT) for the IBM 704. LISP 1.5 became the standard LISP implementation for many years. Today, the best-known general-purpose Lisp dialects are Common Lisp, Scheme, Racket and Clojure.
- COMIT was the first string processing language developed by Dr. Victor Yngve, University of Chicago, and collaborators at MIT from 1957 to 1965.
- SNOBOL (String-Oriented Symbolic Language) was invented in 1962 by Bell Labs. SNOBOL then progressed until it reached it's finally implementation in 1967 called SNOBOL4.
- Prolog (PROgramming in LOGics) was developed in 1972 by Alain Colmerauer with Philippe Roussel, based on concepts from mathematical logic at University of Edinburgh.

Short History of Programming Languages

- **System Languages:**
  - CPL (Combined Programming Language) was developed jointly between the Mathematical Laboratory at the University of Cambridge and the University of London Computer Unit during the 1960s. It is predecessor of the language BCPL, which, in turn, is a precursor of the C language.
  - BCPL (Basic Combined Programming Language) was first implemented by Martin Richards of the University of Cambridge in 1967.
  - Both CPL and BCPL were never widely used. C changed that all. Significant portion of UNIX is written in C language, and only a tiny portion is coded in assembly language for specific hardware.

Short History of Programming Languages

- **Evolution of Software Architectures:**
  - The hardware that supports a language has a great impact on language design. Each era has had a profound effect on the set of languages used.
  - **Mainframe Era:**
    - **Batch environments:** Input data are collected in batches on files and are processed in batches by the program.
    - **Interactive environments:** A program interacts directly with a user at a display console during execution, by alternately sending output to the display and receiving input.
    - **Effects on language design for batch processing:** Files are usually the basis for most of the I/O; Error that terminates execution of the program is acceptable but costly because entire run must be repeated; Language usually provides no facilities for monitoring or directly affecting the speed at which the program executes, that is, lack of timing constraints on a program.

Short History of Programming Languages

  - **Effects on language design for interactive processing:** Include functions for accessing lines of text from a file and other functions that directly input each character as typed by the user at a terminal; The program may display an error message and ask for correction from the user for bad input, termination of the program in response to error is usually not accepted unlike batch processing; An interactive program that operates so slowly that it cannot respond to an input command in a reasonable period is often considered unusable, utilize some notion of timing.

- **Personal Computer Era:**
  - The mainframe time-sharing era was very short-lived, lasting perhaps from the early 1970s to the mid-1980s. The personal computers (PC) changed all that.
  - **Personal Computers:** These computers are microcomputers that use microprocessor and are designed for use by only one person at a time.

Short History of Programming Languages

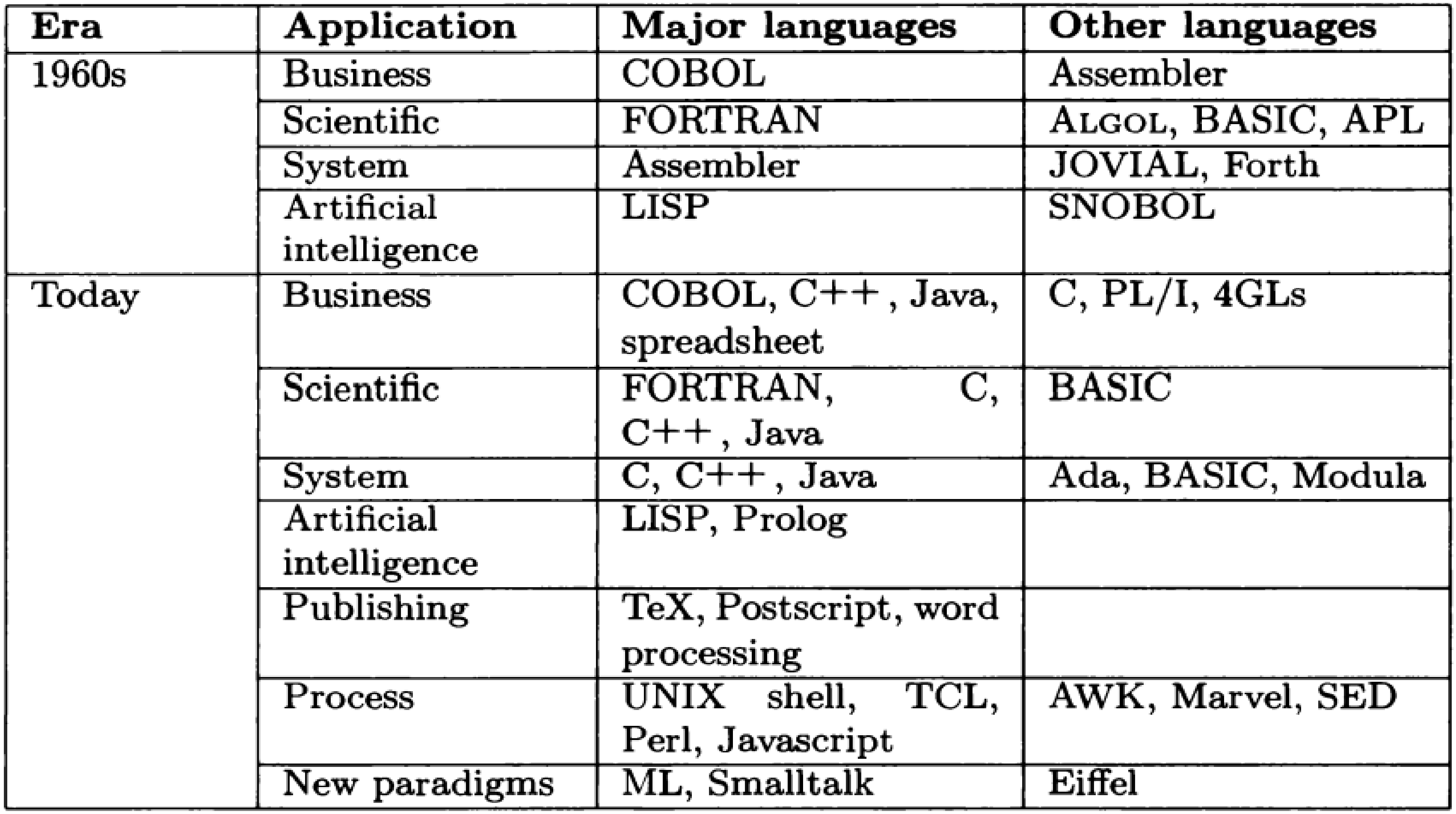
- **Embedded-system environments:** A computer system that is used to control part of a larger system such as an industrial plant, an aircraft, an automobile etc.
- **Effects on language design for personal computers:** With the advent of user interfaces such as windows, developing languages with good interactive graphics becomes of primary importance; Object-oriented programming is a natural model for this environment.
- **Effects on language design for embedded systems:** Programs written for embedded systems often operate without an underlying operating system and without the usual environment of files and I/O devices. The program must interact directly with nonstandard I/O devices. Error handling is of particular importance. Embedded systems almost always operate in real time. An embedded computer system is often a distributed system composed of more than one computer and program is usually composed of a set of tasks that operate concurrently.

Short History of Programming Languages

- **Networking Era:**
  - **Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers or nodes. Even though the software components may be spread out across multiple computers in multiple locations, they're run as one system.
  - **Internet:** A global computer network providing a variety of information and communication facilities, consisting of interconnected networks using standardized communication protocols.
  - **Effects on language design:** Requires the use of languages that allow interaction between the client and server computers. Use of client and server side languages to design Web pages. Use of languages to develop email user agent programs.

Short History of Programming Languages

- **Application Domains:**
  - The appropriate language to use often depends on the application domain for the problem to be solved.



Era	Application	Major languages	Other languages
1960s	Business	COBOL	Assembler
	Scientific	FORTRAN	ALGOL, BASIC, APL
	System	Assembler	JOVIAL, Forth
	Artificial intelligence	LISP	SNOBOL
Today	Business	COBOL, C++ , Java, spreadsheet	C, PL/I, 4GLs
	Scientific	FORTRAN, C, C++ , Java	BASIC
	System	C, C++ , Java	Ada, BASIC, Modula
	Artificial intelligence	LISP, Prolog	
	Publishing	TeX, Postscript, word processing	
	Process	UNIX shell, TCL, Perl, Javascript	AWK, Marvel, SED
	New paradigms	ML, Smalltalk	Eiffel

Attributes of a Good Language

- **Clarity, simplicity, and unity:**
  - A language should provide a clear, simple, and unified set of concepts that can be used as primitives in implementing algorithms.
  - The syntax of a language affect the ease with which a program may be written, tested, and later understood and modified.
- **Orthogonality:**
  - Orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. For example, expression and conditional statements are orthogonal if any expression can be evaluated within a conditional statement.
  - Orthogonal languages are easier to learn and programs are easier to write.

Attributes of a Good Language

- **Naturalness for the application:**
  - It should be possible to translate algorithms directly into appropriate program statements.
  - Languages should provide appropriate data structures, operations, control structures, and a natural syntax for the problem to be solved.
  - Naturalness simplify the creation of programs in particular area. For example, Prolog is used for deductive properties and C++ is used for object-oriented properties.
- **Support for abstraction:**
  - Languages should allow data structures, data types, and operations to be defined and maintained as self-contained abstractions.
  - The programmer may use them in other parts of the program knowing only their abstract properties without concern for the details of the implementation.

Attributes of a Good Language

- **Ease of program verification:**
  - There are many techniques for verifying that a program correctly performs its required function, such as, formal verification method, desk checking, executing, and so on.
  - A language that makes program verification difficult may be far more troublesome to use than one that supports and simplifies verification.
  - Simplicity of semantic and syntactic structure is a primary aspect that tends to simply program verification.
- **Programming environment:**
  - The presence of an appropriate programming environment may make a technically weak language easier to work with.
  - A long list of factors, such as, language editors, language debugging facilities, versioning features etc., might be included as part of the programming environments.

Attributes of a Good Language

- **Portability of programs:**
  - One important criterion for many programming projects is the ***transportability*** of the resulting programs from the computer on which they are developed to other computer systems.
  - A language that is widely available and whose definition is independent of the features of a particular machine forms a useful base for the production of transportable programs.
- **Cost of use:**
  - ***Cost of program execution*** – Cost of program execution is of primary importance for large production programs that will be executed repeatedly.
  - ***Cost of program translation*** – It is important to have a fast and efficient compiler rather than a compiler that produces optimized executable code.

Attributes of a Good Language

- ***Cost of program creation, testing, and use*** – For a certain class of problems, a solution may be designed, coded, tested, modified, and used with a minimum investment of programmer time and energy.
- ***Cost of program maintenance*** – A language that makes it easy for a program to be repeatedly modified, repaired, and extended by different programmers over a period of many years may be, in the long run, much less expensive to use than any other.

Syntax and Semantics

- The **syntax** of a programming language is what the program looks like. To give rules of syntax for a programming language means to tell how statements, declarations, and other language constructs are written. for example,

  int V[10]; is the declaration of 10-element vector, V, of integers and V: array[0..9] of integer; is Pascal declaration.

- The **semantics** of a programming language is the meaning given to the various syntactic constructs.

Language Paradigms

- A **programming paradigm** is a style, or “way,” of programming.
- The programming paradigm is defined as a set of principles, ideas, design concepts and norms that defines the manner in which the program code is written and organized.
- A programming paradigm is a framework that defines how the programmer can conceptualize and model complex problem to be solved.
- There are two main types of programming paradigms: **imperative paradigm** and **declarative paradigm**.
- **Imperative Paradigms:**
    - This paradigm is said to be command driven. The program code in imperative paradigm directs the program execution as sequence of statements execute one by one.

Language Paradigms

- The imperative style program consists of a set of program statements and each statement directs the computer to perform specific task.
- The execution of the program statements is decided by the control flow statements.
- The imperative style of programming focuses on how it is to be done.
- Instead of giving your friend simply your address, you would give the detail step by step to reach your place.
- There are three types of imperative programming: **procedural programming**, **object-oriented programming**, and **database query language**.
- **Procedural programming** consists of set of functions, each performing a specific operation. For example, C programming.

Language Paradigms

- **Object-oriented programming** uses program components represented as objects. Uses different concepts such as, classes and objects, encapsulation and data hiding, inheritance, polymorphism, and abstraction. For example, C++, Java, Python etc.
- **Database query languages** are used to manipulate data stored in a database. For example, SQL.

- **Declarative Paradigms:**
  - This paradigm focuses on the logic of the program and the end result. The main focus is achieving the end result.
  - This approach is pretty much straight forward and focuses on what is to be done.
  - Control flow is not the important element of this paradigm.
  - For example, you simply give your friend your address and let your friend figure out how to reach your home.

Language Paradigms

- There are two types of declarative programming: **functional programming** and **logic programming**.
- **Functional programming** attempts to solve problems by composing mathematical functions as program components. The foundation for functional programming is Lambda Calculus. For example, LISP.
- **Logic programming** is based on logic and control. Logic essentially means facts and rules whereas control means an order of rules. For example, Prolog.

Language Standardization

- A programming language code can be validated and evaluated using three approaches:
  1. Read the definition in the language reference manual typically published by the vendor of your local compiler.
  2. Write a program on your computer to see what happens.
  3. Read the definition in the **language standard**.
- Option 2 is probably the most common. Option 1 can also be checked. Option 3 is rarely employed.
- Option 1 and 2 are tied to a particular implementation. But is that implementation correct? What if you want to move your 50,000-line program to another computer that has a compiler by a different vendor? Will the program still compile correctly and produce the same results when executed? If not, why not? Is this legal to add new features to the language by vendors to enhance its usefulness?

Language Standardization

- To address these concerns, most languages have standard definitions. All implementations should adhere to this standard. Standards generally come in two flavors:
  1. **Proprietary standards.** These are definitions by the company that developed and owns the language. These standards do not work for languages that have become popular and widely used. Soon come with many enhancements and incompatibilities.
  2. **Consensus standards** (or simply **standards**)**.** These are documents produced by organizations based on an agreement by the relevant participants. These are major methods to ensure uniformity among several implementations of a language.

Language Standardization

- Each country typically has one or more organizations assigned with the role of developing standards. For example, American National Standards Institute (ANSI), Computer Business Equipment Manufacturers Association (CBEMA), Institute of Electrical and Electronic Engineers (IEEE), British Standards Institute (BSI), International Standards Organization (ISO), National Institute of Standards and Technology (NIST) etc.
- Three issues need to be addressed to use standards effectively:
  1. **Timeliness.** When do we standardize a language? One would like to standardize a language early enough so that there is enough experience in using the language, yet not so late as to encourage many incompatible implementations.

Language Standardization

2. **Conformance.** What does it mean for a program to adhere to a standard and for a compiler to compile a standard? A program is conformant if it only uses features defined in the standard. A conforming compiler is one that, when given a conformant program, produces an executable program that produces the correct output.
3. **Obsolescence.** When does a standard age, and how does it get modified? As our knowledge and experience of programming evolve, new computer architectures require new language features. Once we standardize a language, it seems quaint a few years later.

Language Internationalization

- With the globalization, programming is increasingly a global activity, and it is important for languages to be readily usable in multiple countries.
- There is increasing need for computers to “speak” many different languages. For example, use of an 8-bit, can store up to 256 different character representations, to represent character is often insufficient.
- Issues such as character codes, collating sequences (sorting, case, and scanning direction), formats for date and time, ideographs, currency etc. affect input and output data.

Programming Environments

- A programming environment is the environment in which programs are created and tested.
- It tends to have less influence on language design than the operating environment in which programs are expected to be executed.
- A programming environment consists primarily of a set of *support tools* and a *command language* for invoking them.
- Each support tool is another program that may be used by the programmer as an aid during one or more of the stages of creation a program.
- Typical tools in a programming environment include *editors*, *debuggers*, *verifiers*, *test data generators*, and *pretty printers*.

Programming Environments

- **Effects on Language Design:**
  - Programming environments have affected language design primarily in two major areas: features aiding **separate compilation** and assembly of program from components, and features aiding program **testing and debugging**.
  - **Separate compilation:**
    - This feature requires the language to be structured so that individual subprograms or other parts can be separately compiled and executed, without the other parts, and then later merged without change into the final program.
    - Separate compilation is made difficult by the fact that in compiling one subprogram, the compiler may need information about other subprograms or shared data objects.

Programming Environments

- To provide information about separately compiled subprograms, shared data, and type definitions either (1) the language may require that information may be redeclared within the subprogram; (2) it may prescribe a particular order of compilation to require compilation of each subprogram to be preceded by compilation of the specification of all called subprograms and shared data; or (3) it may require the presence of a library containing the relevant specifications during compilation so that the compiler may retrieve them as needed.
- Option 1, also called *independent compilation,* has disadvantage that if the declarations within the subprogram do not match the actual structure of the data or subprogram, then subtle error appears in the final assembly stage that will not be detected during testing of the independently complied program parts.
- Option 2 and 3 require a means for specifications of subprograms, type definitions, and common environments to be given or placed in a library prior to the compilation of a subprogram.

Programming Environments

  - Separate compilation also affects language design in the use of shared names. If several groups are writing portions of a large program, it is often difficult to ensure that the names used by each group are distinct. A common problem is to find same names of subprograms and other program units during assembly of the final complete program. Languages employ three methods to avoid this problem: (1) Each shared name must be unique and naming conventions must be adopted, (2) Languages often use scoping rules to hide names, and (3) Names may be known by explicitly adding their definitions from an external library, such as, inheritance in object-oriented languages.

- **Testing and Debugging:**
  - Most languages contain some features to aid program testing and debugging. Some examples are given below:
  - **Execution trace features.** Many interactive languages provide features that allow particular statements and variables to be tagged for tracing during execution.

Programming Environments

- **Breakpoints.** Interactive languages often provide a feature where the programmer can specify points in the program as breakpoints. When a breakpoint is reached during execution, execution of the program is interrupted, and control is given to the programmer at a terminal. The programmer may inspect and modify values of variables and then restart the program from the point of interruption.
- **Assertions.** It is a conditional expression inserted as a separate statement in a program. The assertion states the relationships that must hold among the values of the variables at that point in the program. During program execution, if the conditions fail to hold, then execution is interrupted, and an exception handler is invoked to print a message or take other action. Assertions may be enabled or disabled in the program. When disabled, they become comments that aid in documenting the program.

Programming Environments

- **Environment Framework:**
  - Environment framework supplies services such as *data repository*, *graphical user interface*, *security*, and *communication services*.
  - Programs are written to use these services.
  - Languages are sometimes designed to allow for easy access to these infrastructure services.
- **Job Control and Process Languages:**
  - Related to environment frameworks is the concept of job control.
  - Job Control Language(JCL) is a scripting language that describe jobs, to the Operating System that runs in the IBM large server (Mainframe) computers.
  - JCL acts as an interface between your programs (COBOL, PL/1 , Assembler etc.) and Mainframe OS (MVS or Z/OS).

Programming Environments

- It is mainly a set of control statements that provide the specifications necessary to run an application program. In mainframe environment, programs can be executed in batch and online modes. JCL is used for submitting a program for execution in batch mode.
- For every job that you submit, you need to tell the OS where to find the appropriate input, how to process that input, and what to do with the resulting output. JCL is used to convey this information to OS.
- All jobs require the three main types of JCL statements: **JOB** (indicates job related information such as job id, priority, user id), **EXEC** (indicates name of the program to be executed), and **DD** (data descriptor indicates data that need to be processed or produced by the program).