Unit 1

Programming Language Syntax and semantics
1.1 Software development process

- The software is said to have a *life cycle composed* of several phases.
- Each of these phases results in the development of either a part of the system or something associated with the system, such as a fragment of specification, a test plan or a users manual.
- In the traditional *waterfall model of the software life cycle*, the *development process is a* sequential combination of phases, each having well-identified starting and ending points, with clearly identifiable deliverables to the next phase.
- Each step may identify deficiencies in the previous one, which then must be repeated.
A sample software development process based on the waterfall model may be comprised of the following phases:

1. **Requirement analysis and specification** - The result of this phase is a requirements document stating what the system should do, along with users' manuals, feasibility and cost studies, performance requirements, and so on.

2. **Software design and specification** - The result of this phase is a system design specification document identifying all of the modules comprising the system and their interfaces.

3. **Implementation (coding)** - The system is implemented to meet the design specified in the previous phase.

4. **Verification and validation** - Two specific kinds of assessment performed during implementation are *module testing* and *integration testing*.

5. **Maintenance** - changes to the system may become necessary either because of detected malfunctions, or a desire to add new capabilities or to improve old ones, or changes that occurred in operational environment.
1.2 Languages and software development environments

• We will examine the role of the programming language in the software development process by illustrating the relationship between the programming language and other software development tools in this Section.

• The work in any of the phases of software development may be supported by computer-aided tools.
  Eg: text editors, compilers, linkers, and libraries, debugger (used to locate faults in a program and eliminate them).

• By a software development environment we mean an integrated set of tools and techniques that aids in the development of software.

• A team of application and computer specialists interacting with the environment develops the system requirements. The environment keeps track of the requirements as they are being developed and updated, and guards against incompleteness or inconsistency.
• The environment provides support for these phases by automating some development steps, by suggesting reuse of existing design and implementation components taken from a library, by recording the relationships among all of the artifacts, so that one can trace the effect of a change in—say—the requirements document to changes in the design document and in the code.

• The tools include: programming language processors, such as editors, compilers, simulators, interpreters, linkers, debuggers, and others.

• All of the tools must be compatible and integrated with tools used in the other phases.
1.3 Languages and software design methods

• We will illustrate the relationship between the programming language and design methods in this Section.
• The relationship between software design methods and programming languages is an important one. Some languages provide better support for some design methods than others.
• Older languages, such as FORTRAN, were not designed to support specific design methods. For example, the absence of suitable high-level control structures in early FORTRAN makes it difficult to systematically design algorithms in a top-down fashion.
• Programming languages may enforce a certain programming style, often called a *programming paradigm*.
• For example, Smalltalk and Eiffel enforce the development of programs based on object classes as the unit of modularization, as object oriented programming languages.
• Languages enforcing a specific programming paradigm can be called paradigm-oriented.
• There need not be a one-to-one relationship between paradigms and programming languages. Some languages, in fact, are paradigm-neutral and support different paradigms. For example, C++.
• Design methods, in turn, guide software designers in a system’s decomposition into logical components which, eventually, must be coded in a language.
• If the design method and the language paradigm are the same, or the language is paradigm-neutral, then the design abstractions can be directly mapped into program components. Otherwise, if the two clash, the programming effort increases.
• Here we review the most prominent programming language paradigms, with special emphasis on the unit of modularization promoted by the paradigm.
1. **Procedural programming** - Procedures and functions

2. **Functional programming** - It emphasizes the use of expressions and functions

3. **Abstract data type programming** - abstract data types as the unit of program modularity, CLU was the first language designed specifically to support this style of programming

4. **Module-based programming** - emphasizes modularization units that are groupings of entities such as variables, procedures, functions, types, etc.

5. **Object-oriented programming**

6. **Generic programming** - This style emphasize the definition of generic modules that may be instantiated, either at compile-time or runtime, to create the entities—data structures, functions, and procedures

7. **Declarative programming** - This style emphasizes the declarative description of a problem. Eg- Logic languages, like PROLOG, and rule-based languages, like OPS5 and KEE,
1.4 Languages and computer architecture

- Design methods influence programming languages in the sense of establishing requirements for the language to meet in order to better support software development.
- Languages have been constrained by the ideas of Von Neumann, because most current computers are similar to the original Von Neumann architecture (Fig. 1.1).
- Conventional programming languages can be viewed as abstractions of an underlying Von Neumann architecture. For this reason, they are called Von Neumann languages. An abstraction of a phenomenon is a model which ignores irrelevant details and highlights the relevant aspects.
- Conventional programming languages keep their computation model from the underlying Von Neumann architecture, but abstract away from the details of the individual steps of execution.
- Conventional languages based on the Von Neumann computation model are often called imperative languages. Other common terms are state-based languages, or statement-based languages, or simply Von Neumann languages.
FIGURE 2. Requirements and constraints on a language.
1.5 Programming language qualities

• **Software must be reliable** (the chance of failures due to faults in the program should be low.)

• **Software must be maintainable** (economic considerations have reduced the possibility of throwing away existing software and developing similar applications from scratch. Existing software must be modified to meet new requirements.)

• **Software must execute efficiently** (Although the cost of hardware continues to drop as its performance continues to increase (in terms of both speed and space))

These three requirements—reliability, maintainability, and efficiency—can be achieved by adopting suitable methods during software development, appropriate tools in the software development environment, and by certain characteristics of the programming language.
a. Languages and Reliability

- Writable (problem solving activity and should not distracted by language details)
- Readability (Use of English text and familiar notations)
- Simplicity (simplicity of algorithm, more simple the program is less powerful)
- Safety (avoid some program constraints that are harmful in program)
- Robustness (ability to deal with undesired events in program, system can be predictable in undesired situation if events are trapped and suitable responses are programmed)
b. Languages and Maintainability

System should be easy to modify, modification are expected due to changes in technology or new requirement from customer

- Language Features used to provide maintainability are
  - Factoring - language should allow programmers to factor related features into one single point
  
  Eg: i) identical operation repeated in several points can be factored in an routine and routine calls can be used, further changes will be done in routine only and that will reflect everywhere

  ii) Instead of constants we can define a name for constant and can change definition of constant only if required, example – we can use pi instead of 3.14

  - Locality – expected changes in program should be concentrated over small area of program, otherwise if it extends to large part of program then it becomes more complex.

  Eg: Abstract data type use will be easy to modify if ADT manipulation is invoked same way.
c. Languages and Efficiency

• Many languages have had efficiency as a main design goal, either implicitly or explicitly. For example, FORTRAN originally was designed for a specific machine (the IBM 704). Many of FORTRAN's restrictions, such as the number of array dimensions or the form of expressions used as array indices, were based directly on what could be implemented efficiently on the IBM 704.

• Efficiency is measured in terms of speed, space, initial efforts required to design system, and maintenance efforts.

• Productivity or efficiency of software development process rather than performance of the resulting product can be improved in terms of Reusability
  Portability
  Code Optimization by compiler
  Concurrent programming
  (Free) Memory reuse
1.6 A brief historical perspective

- Early time programming was used by *single user, scientist, researchers, without any requirement of analysis and maintenance*
- The *desire to apply the computer in more and more applications* led to its being used in increasingly less understood and more sophisticated environments. This, in turn, led to the *need for “teams” of programmers* and more disciplined approaches.
- High Level Language- languages with high level of abstraction
- Because so much effort and money was being spent on the development of systems, *old systems could not simply be thrown away* when a new system was needed. *Economic considerations forced people to enhance an existing system* to meet the newly recognized needs.
- *System reliability* is another issue that has gained importance gradually, because of two major factors. One factor is that systems are being developed for *users with little or no computer background*; The second factor is that systems are now being applied in *critical areas such as chemical or nuclear plants and patient monitoring*. In order to ensure reliability, *verification and validation* became vital.
<table>
<thead>
<tr>
<th>Language</th>
<th>Year</th>
<th>Originator</th>
<th>Intended Purpose</th>
<th>Predecessor Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>1954-57</td>
<td>J. Backus</td>
<td>Numeric computing</td>
<td></td>
</tr>
<tr>
<td>ALGOL 60</td>
<td>1958-60</td>
<td>Committee</td>
<td>Numeric Computing</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>LISP</td>
<td>1956-62</td>
<td>J. McCarthy</td>
<td>Symbolic Computing</td>
<td></td>
</tr>
<tr>
<td>PL/I</td>
<td>1963-64</td>
<td>Committee</td>
<td>General Purpose</td>
<td>FORTRAN, ALGOL 60, COBOL</td>
</tr>
<tr>
<td>SIMULA 67</td>
<td>1967</td>
<td>O.-J.Dahl</td>
<td>Simulation</td>
<td>ALGOL 60</td>
</tr>
<tr>
<td>Pascal</td>
<td>1971</td>
<td>N. Wirth</td>
<td>Educational and gen. purpose</td>
<td>ALGOL 60</td>
</tr>
<tr>
<td>C</td>
<td>1974</td>
<td>D. Ritchie</td>
<td>Systems Programming</td>
<td>ALGOL 68</td>
</tr>
<tr>
<td>Concurrent Pascal</td>
<td>1975</td>
<td>P. Brinch Hansen</td>
<td>Concurrent Programming</td>
<td>Pascal</td>
</tr>
<tr>
<td>CLU</td>
<td>1974-77</td>
<td>B. Liskov</td>
<td>ADT programming</td>
<td>SIMULA 67</td>
</tr>
<tr>
<td>Modula-2</td>
<td>1977</td>
<td>N. Wirth</td>
<td>Systems Programming</td>
<td>Pascal</td>
</tr>
<tr>
<td>Language</td>
<td>Year</td>
<td>Originator</td>
<td>Intended Purpose</td>
<td>Predecessor Language</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>-------------------</td>
<td>--------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>Ada</td>
<td>1979</td>
<td>J. Ichbiah</td>
<td>General purpose Embedded Systems</td>
<td>Pascal, SIMULA 67</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>1971-80</td>
<td>A. Kay</td>
<td>Personal Computing</td>
<td>SIMULA 67, LISP</td>
</tr>
<tr>
<td>C++</td>
<td>1984</td>
<td>B. Stroustrup</td>
<td>General purpose</td>
<td>C, SIMULA 67</td>
</tr>
<tr>
<td>KEE</td>
<td>1984</td>
<td>Intellicorp</td>
<td>Expert systems</td>
<td>LISP</td>
</tr>
<tr>
<td>Linda</td>
<td>1986</td>
<td>D. Gelernter</td>
<td>Parallel/ distributed Programming</td>
<td></td>
</tr>
<tr>
<td>TCL/TK</td>
<td>1988</td>
<td>J. K. Ousterhout</td>
<td>Rapid development, GUIs</td>
<td>OS shell languages</td>
</tr>
<tr>
<td>Java</td>
<td>1995</td>
<td>SUN Microsystems</td>
<td>Network Computing</td>
<td>C++</td>
</tr>
</tbody>
</table>
1.7 A bird’s eye view of programming language concepts

• This point provides a bird’s eye view of the main concepts of programming Languages to show how all the various concepts that will be presented fit together in a coherent picture. Using a simple C/C++ program as an example, we look at the kinds of facilities that a programming language must support and the different ways that languages go about providing these facilities.

1) A Simple Program
• We are interested in the kinds of things one can do with programming languages, rather than the specifics of a given program.
• What are the inherent capabilities and shortcomings of different programming languages?
• What makes one language fundamentally different from another and what makes one language similar to another, despite apparent differences?
Any program mainly consist of three sections
• Structure (header file, namespace, library files)
• Environment (variables, functions declared before main)
• Computation (Combination of algorithms and program’s data)

2) Syntax and Semantics

Syntax and Semantics determine validity of program
• Any programming language specifies a set of rules for the form of valid programs in that language.
• The syntax rules of the language state how to form expressions, statements, and programs that look right.
Eg: variable must be declared before it is used in statement
• The semantic rules of the language tell us how to build meaningful expressions, statements, and programs.
Eg: Characters are the ultimate syntactic building blocks. Every program is formed by placing characters together in some well-defined order.
3) Semantic elements

In this section we will examine basic semantic elements of programming language from the language designer’s point of view. We can examine what choices may be available to a language designer and how the designer’s decisions affect the programmer.

i) Variable: A variable is the most pervasive concept in traditional programming languages. A variable corresponds to a region of memory which is used to hold values that are manipulated by the program. We refer to a variable by its name.

The syntactic rules specify how variables may be named.

A declaration introduces a variable by giving it a name and stating some of its semantic properties. The important semantic properties are:

- Scope
- Type
- Lifetime
ii) Values and references:

what is the value associated with a variable?

*Ans: Content and Reference*

*Eg:* \( x = y; \)

L-value: a value that refers to a memory location

R-value: a value that refers to the contents of a memory location

Other example:

\[
\begin{align*}
    x &= &\& y; \\
    y &= 3;
\end{align*}
\]

- Some contexts require a particular type of value. For example, the left-hand side of an assignment statement requires an l-value.

  therefore \( 3 = y \) is error, as left hand side require l-value
iii) **Expression**: Expressions are syntactic constructs that allow the programmer to combine values and operations to compute new values. The language specifies syntactic as well as semantic rules for building expressions.

- Expression may generate single type of value or of different types

Eg: expression of type boolean or of type string
4) Program organization

Number of lines → Modules → Files

Each file contains number of modules, each module contains number of lines of code

Each file is prepared for execution for some specific functionality

Advantages of Such program organization:

- Each file can be developed by different developer ultimately increase efficiency
- Each file can be easily maintained
- Requirement to make Changes to any module does not disturb any other module
- Bug finding and correcting is easy and less complex
5) Program data and algorithms:
Programs implement algorithms – algorithms operate on some data or variables to produce some result

There are some kind of values that a variable can hold.

• Where can such a variable declaration occur in a program? Only at the beginning of a program or anywhere? When is the variable created? Does it have an initial value? Is it known to other procedures or modules of the program? How can variables be exported to other modules? What are the kinds of components that a data structure may contain? Can a function be an element of a record?

• Sophisticated mechanisms for data definition allow the programmer to modularize the data in the program similarly to the way that the algorithms are modularized.

Computations: control structures, routines

• Combining expressions, statements, control structures and routine calls in C++ and other conventional languages allows the programmer to write algorithms using an imperative computation paradigm.
6) External Environment:

Library

Header files

Interface to interact with external environment

Files can be used to transfer data

Constructs for input/output

• The advantage of language-supported facilities for communication with the external environment is that the programmer has a complete model of the environment and the compiler can do consistency checking.

• Supporting the facilities in a library makes the language simpler and allows more flexibility. For example, different libraries may be added as new devices, such as graphical ones, become available.