1.1 INTRODUCTION TO PRINCIPLES OF PROGRAMMING LANGUAGES

Software designs are to be implemented in any one of the programming languages. For efficient implementation of any software design, the knowledge of programming languages is essential. System analysts, software engineers, programmers can perform their task effectively if they studied principles of programming languages.

Programming languages are notations. They are used for specifying, organizing and reasoning about computations. Programming languages are specified by rules for forming correct statements, organizing them into modules, submitting them to a compiler, which translates the code into a language understandable by a particular machine and finally running the program, i.e., submitting input to the computer, which transforms it into output according to the instructions in the program.

A programming language is a systematic notation by which we describe computational processes to others. Computational process means a set of steps which a machine can perform for solving a task.

1.1.1 Different Levels of Programming Language

The computer describes the solution of a problem, we need to know about the set of commands that the computer can understand and execute. Therefore, the language is needed that the computer can understand.

The different levels of programming languages are as follows,

(1) Machine language.

(2) Assembly language.

(3) High level language.

(4) 4GL language.

Fig. 1.1.1 shows the different levels of programming languages.

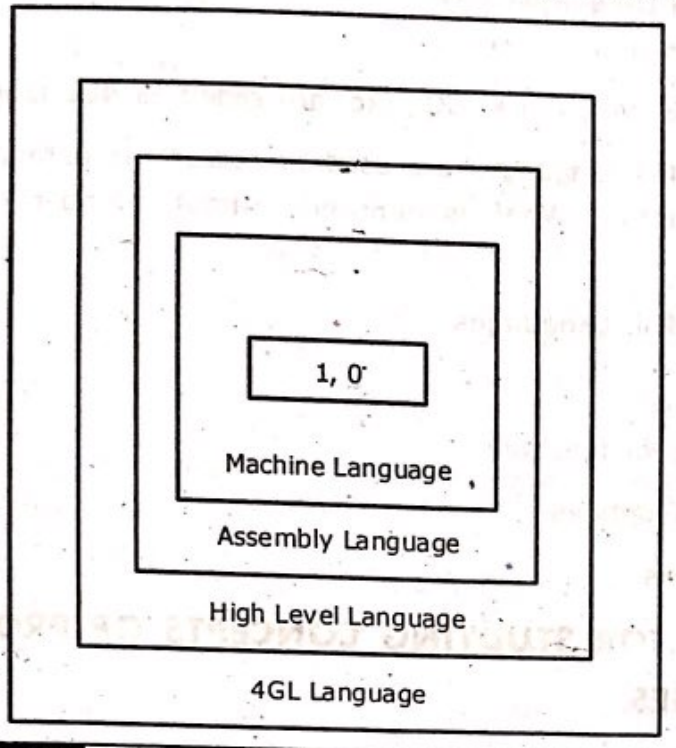


Fig. 1.1.1 Different Levels of Programming Languages

(1) **Machine Level Language :** Previously, this language was used to program a computer. In machine language programs are written using 0's and 1's. This language is uses very few programs for mainly two reasons,

(i) It is very difficult to work.

(ii) Every different type of computer has its own set of instructions.

(2) **Assembly Level Language :** In this assembly level language only names and symbols are used when names and symbols are used, when compare to the machine level language it is easy to write the program in the assembly language. Assembler is used to convert the programs from assembly to machine level languages.

(3) **High Level Language :** The high level language has replaced machine and assembly languages in all areas of programming because they provide benefits like,

(i) Readable familiar notations.

(ii) Availability of program libraries.

(iii) Machine independence.

(iv) Consistency checks during implementation that detect the errors.

1.3 PROGRAMMING DOMAINS

The appropriate language to use often depends on the application domain for the problem to be solved since different languages are defined for taking care of various computer applications.

1.3.1 Scientific Applications

Scientific programming is primarily concerned with making complex calculations very fast and accurate. The calculations are defined by mathematical models that represent scientific phenomena. They are primarily implemented using the imperative programming paradigm. FORTRAN was the first higher level language used for scientific applications. The syntax of FORTRAN is similar to mathematics and the scientists find it easy to use.

1.3.2 Business Applications

Business applications refers to the large processing applications such as order-entry programs, inventory control and payroll. The language used to develop these applications is COBOL. It is still used in data processing applications, but after the invention of 4 GLS, database systems and spread sheets are most widely used.

1.3.3 Artificial Intelligence

The Artificial Intelligence (AI) programming community which is responsible for developing programs that can model human intelligent behaviour and logical deductions has been under working for a long period of time. The activities such as symbol manipulations, functional expressions and the logical proof systems design are the essential requirements for artificial intelligence applications. Functional and logic programming results from artificial intelligence applications. The first functional programming language developed for AI applications was LISP and then the PROLOG logic programming was invented.

1.3.4 Systems Programming

Systems programming refers to the activity of designing and maintaining the basic software that runs the components of operating system, network software, language compilers and debuggers, virtual machines and interpreters etc. The people involved in this activity are called systems programmers. These programs can be either written in an assembly language or in some specific language of systems programming. Systems programming can be used along with imperative event-driven paradigm. C is the most widely used systems programming language.

Key Features Provided by C Language which Makes it Suitable for System Programming : As C is a low-level language with low-level features, it is more appropriate for system programming domain. It is an implementation language developed by Dennis Ritchie and is used in writing UNIX operating system. It is associated with UNIX operating system because of its portability.

C is embedded with various features that makes it suitable for system programming. They are as follows,

(1) C is a compiled language which makes execution efficient. As the speed of execution plays a vital role in systems programming, C is commonly used.

(2) C is associated with a rich set of operators that provides a high degree of expressiveness.

(3) The control statements and data structuring facilities of C makes it suitable for many diverse application areas.

(4) The facility of pointers in C solves many complex problems especially in the systems programming environment.

(5) A language implementation system requires many-facilities of an operating system. So, the operating system and language implementation are built over the machine language interface of a computer. Lower layer acts as a virtual computer and provides an interface to the higher layer. This virtual computer is provided by the C compiler and the operating system.

(6) Explicit storage allocation and deallocation is done in C-language.

(7) Use of bit-wise operators of C in systems programming saves a few nanoseconds of run-time.

(8) The syntax associated with C language makes its usage simple and easy.

(9) C doesn't imposes safety restrictions hence, reduces the burden of the users.

(10) Type conversions in C-language prevents the programmers from explicitly connecting intermediate values in many arithmetic calculations.

1.3.5 Scripting Languages

A scripting language is used by placing a list of commands called a script, in some file and then they are executed. 'Sh' was the first language for this domain, 'Sh' consists of a small collection of commands that can be interpreted as calls to the subprograms, these subprograms can perform utility functions such as file management and file filtering. Hence, many control flow statements, functions, variables and other capabilities were added to obtain a complete programming language. Due to the invention of WWW many scripting languages were introduced in the recent past. Among them was JavaScript, VB script, PERL and Common Gateway Interface (CGI).

1.3.6 Special-Purpose Languages

These languages are designed for a specific problem domain and have narrow applicability. These languages are ranging from RPG(A language used to generate business reports) to APT(A language used for instructing programmable machine tools) and to GPSS (A language used for systems simulation). The problem with these languages is that they are difficult to compare with other programming languages.

PROLOG is the most widely used special purpose language which was designed to serve the narrow range of natural language processing.

1.4 LANGUAGE EVALUATION CRITERIA

The concepts of various constructs and capabilities of programming languages is to examine carefully. We also evaluate these features, focusing on their impact of the software development process, including maintenance. So we have need a set of evaluation criteria.

A Programming language possess many characteristics. These characteristics or features can be evaluated based on the language evaluation criteria. Table 1.4.1 shows the language evaluation criteria and the characteristics that effect them.

Table 1.4.1 Characteristics that Affects Evaluation Criteria

S.No.	Characteristics	Criteria		
		Readability	Writability	Reliability
(1)	Simplicity/Orthogonality	Yes	Yes	Yes
(2)	Data types and Structures	Yes	Yes	Yes
(3)	Control Structures	Yes	Yes	Yes
(4)	Syntax Design	Yes	Yes	Yes
(5)	Type Checking	No	No	Yes
(6)	Excepting Handling	No	No	Yes
(7)	Anti Aliasing	No	No	Yes
(8)	Abstraction Support	No	Yes	Yes

1.4.1 Readability

It is the most important criteria for assessing a language i.e., how well it can be easily read and understood. The fundamental characteristics of programming languages are its efficiency and machine readability.

1.4.1.1 Simplicity

The languages readability is greatly influenced by its simplicity. A language that contains a large number of fundamental components is difficult to learn than the language containing small number of fundamental components. Hence programmers who uses a large language must learn only a subset of the language, thereby neglecting all other characteristics,

(1) 'Multiplicity' which signifies the existence of many ways to perform a particular operation also affects readability.

Example : In Java, user can increment a simple integer variable in four different ways,

```
i = i + 1
i + +
+ + i
i + = 1
```

(2) Operator overloading in which a single operator specifies more than one meaning also reduces readability if the users are permitted to construct their own operator overloading.

(3) Most of the assembly language statements are simple, making the (assembly language) programs less readable.

1.4.1.2 Orthogonality

Orthogonality refers to combining various features of a language in all possible ways where each combination gives some well-defined meaning.

A language with orthogonal features is easier to learn and provides an ease of writing the programs. The drawback of orthogonality is that a program will compile without any errors, eventhough many logically incoherent features exists in a program. Thus, many primitive constructs can be combined in a number of combinations that can create, control data structures for the language.

1.4.1.3 Control Statements

Unnecessary use of goto statements greatly reduces programs readability. Reading and understanding a program from top to bottom is easier than jumping from one statement to the other nonadjacent statement. Hence the use of goto statement must be restricted in order to make a program more readable.

It can be restricted in the following ways,

(1) The GOTO statement must always occur before their targets except in the case whe they are used to form loops.

(2) The number of goto statements must be limited.

(3) The targets of the goto statements must not be too far.

BASIC and FORTRAN in early 1970s doesn't have control statements. Most of th programming languages today have sufficient control statements hence, the need for th goto statement is reduced.

1.4.1.4 Datatypes and Structures

Language's readability is also affected by the datatypes and data structures. language which doesn't support boolean data type uses a numeric type to represen a flag as follows.

For a language that doesn't support 'bool' datatype,

```
sum_is_small = 1
```

For a language that supports 'bool' datatype,

```
sum_is_small = true
```

Similarly, 'record' datatypes provide a method for representing records that are mor readable than the collection of similar arrays.

1.4.1.5 Syntax Design

Three syntactic design choices that have a significant effect on readability are as follows,

(1) **Identifier Forms :** Identifiers, when confined to short lengths reduces the language's readability. If the maximum length of an identifier is six characters, then it is impossible to use connotative names for the variables.

Example : ANSI BASIC in which an identifier comprises of a single letter or a single letter along with a single digit.

(2) **Special Words :** Program's appearance and readability can be affected by the special words used in a language especially the method of making compound statements and statement groups etc.

Example : PASCAL, uses begin_end pairs for forming the groups of all control constructs except repeat statement. C uses opening and closing braces for the same purpose.

When the special words are used as names of program variables, then the programs will be very confusing.

(3) **Form and Meaning :** The program's readability can be enhanced by forming the statements in such a way that their appearance specifies their purpose. Language semantics must follow its syntax but, this rule is violated when two constructs have similar appearance with different meanings.

Example : The keyword static in C language serves different meanings in different contexts. If it is associated with a variable inside a function, it indicates that the variable is created at the compile time else if it is used for a variable defined outside all functions then it indicates that the variable can be seen only within the file in which it is defined.

1.4.2 Writability

Writability refers to the ease with which a language can be used to create programs. The characteristics that affect readability also affect writability. The process of writing a program involves rereading the part of the program that is already written. The writability of two languages can't be compared in the realm of a particular application when one of them was designed for that application and the other was not.

Example : The writabilities of COBOL and FORTRAN are different for creating a program to deal with two-dimensional arrays, for which FORTRAN is ideal.

Following characteristics are the most important for writability of a language,

(1) Simplicity and orthogonality.

(2) Support for abstraction.

(3) Expressivity.

1.4.2.1 Simplicity and Orthogonality

If a programming language consists of a large number of constructs then some of the programmers may not be well aware of all these constructs. This may lead to the misuse of some of the language features and many of the other features are not used which may be efficient or elegant. Having a small number of primitive constructs with a set of rules is more efficient than having large primitives. A complex problem can be solved by using a simple set of primitive constructs.

Too much orthogonality can also affects writability.

1.4.2.2 Support for Abstraction

Abstraction refers to the process of defining and using complicated structures or operations in such a way that many of the implementation details are ignored. Hence, the degree of abstraction and the naturalness of its expression greatly affects writability. An example of abstraction is the repeated use of a subprogram for implementing a sort algorithm.

1.4.2.3 Expressivity

Language expressivity refers to many different characteristics. In APL, it means the presence of many powerful operators that deals with large number of computations that can be achieved with a very small program. In C, the representation count ++ is more convenient and shorter than count=count +1. All these increases the writability of a language.

1.4.3 Reliability

A program is said to be reliable if it is working according to its specifications under all conditions. The reliability of a language can be affected by certain features. They are,

(1) Type checking.

(2) Exception handling.

(3) Aliasing.

(4) Readability and writability.

1.4.3.1 Type Checking

It is a process of checking a program for type errors occurred during compilation and program execution. It has a significant effect on reliability. Compile time type checking is desirable because runtime type checking is expensive.

One of the example of how failure is happen to typecheck. In C language the actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function.

An int type variable could be used as an actual parameter in a call to a function that expected a float type as its formal parameter and neither the compiler nor the run-time system would have detected the inconsistency.

1.4.3.2 Exception Handling

Exception handling refers to the ability of a program to detect unusual conditions and run-time errors, take appropriate actions and continue the normal program execution. It affects reliability. The languages such as Ada, C++ and Java have extensive exception handling capabilities.

1.4.3.3 Aliasing

Aliasing is a technique in which a single memory cell is defined by two or more discrete referencing methods or names. It is a dangerous feature in a programming language. Use of aliasing helps in eliminating and controlling the limitations in the data abstraction facilities of a language. Reliability of a language can be increased by restricting the aliasing.

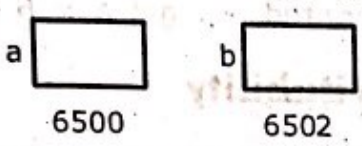
Aliasing deters reliability because altering one value changes the values of all aliases. Moreover, it permits only explicit changes not the implicit changes.

Following example illustrates how aliasing affects reliability,

```
int *a, *b;
a = (int *) malloc (sizeof(int));
*a = 4;
b = a;
*b = 6
printf("%d", *a);
```

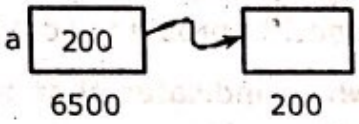
Let us understand the above example,

In line 1, two integer pointer variables 'a' and 'b' are declared as,



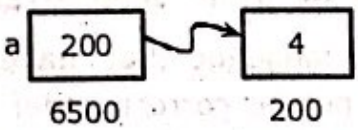
(a)

In line 2, the address which is allocated dynamically is assigned to 'a'. Now, 'a' stores the memory address, say, 200.



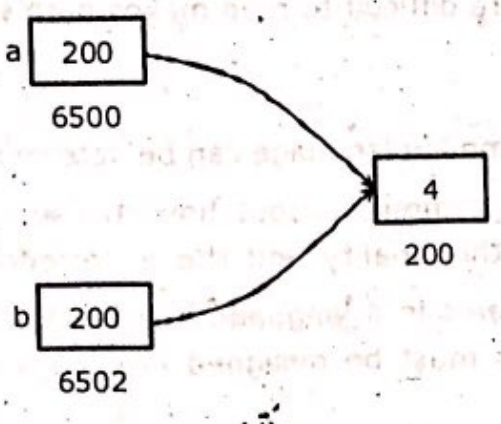
(b)

In line 3, the value of address contained in 'a' is assigned a value 4.



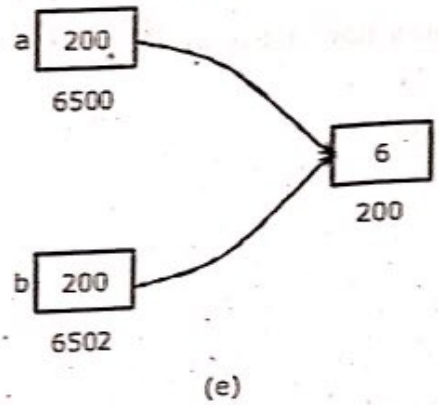
(c)

In line 4, 'a' is copied to 'b'. Now, the value at a (i.e., *a) and the value at b (i.e., *b) are aliases of each other.



(d)

In line 5, the value at address contained in 'b' is assigned a value 6. This statement also changes the value of 'a' to 6.



(e)

Fig. 1.4.1 Process of Aliasing

In line 6, the value of 'a' is printed, we get a = 6 instead of a = 4 due to aliasing.

1.4.3.4 Readability and Writability

Readability refers to how well a language can be easily read and understood. It is a measure of the quality of programs and languages. By the logic of a program, errors can be detected. It is also a subjective criterion. Writability refers to the ease with which a language can be used to create programs. It is a measure of how a language can be used to develop programs for a specific problem domain.

It is a subjective criterion which indicates that higher-level languages are more writable than lower-level languages.

Reliability is the ability of a program to work under all conditions as per its specification. Both readability and writability affects reliability.

Example : A program written in a language that makes use of unusual approaches for implementing the algorithms will not be correct under all possible situations. So, for a program to be reliable, it must be written in a simple and easy manner that can produce the desired and accurate result.

The writing and maintenance phases of reliability are affected by readability. Moreover, the programs that are difficult to read makes both writing and modifying, difficult.

1.4.4 Cost

The total cost of a programming language can be determined from many characteristics.

(1) The cost of training programmers about how to use the language is a function of language's simplicity, orthogonality and the experience of the programmers.

(2) The cost of writing programs in a language is a function of the language's writability. The high-level languages must be designed in such a way that it reduces the cost of creating the software.

(3) The cost of compiling programs in a language.

(4) The cost of executing the programs in a language. This cost is greatly affected by the language's design. A language which involves many run-time type checks will prevent the fast execution of code irrespective of the compiler's quality. Optimization is the technique in which a collection of methods are used by the compilers to reduce the size and increase the execution speed of the code.

(5) The cost of language implementation system. A language whose implementation system is costly or runs only on expensive hardware will not be widely used.

(6) The cost of poor reliability, failure of the software in critical systems results in huge costs.

(7) The cost of maintaining the programs. This cost involves both corrections and modifications to add new capabilities. It mainly depends on readability.

The above mentioned cost contributors, the program development cost, maintenance cost and reliability are most important. Hence, the total cost can be summarized as follows,

Total cost = Cost of training programmers + Cost of writing programs + Cost of compiling programs + Cost of executing programs + Poor reliability cost + Cost of maintaining software + Cost of language implementation system.

A program is said to be reliable if it is working according to its specifications under all conditions. The cost of execution is the cost incurred in executing the programs. If the software fails in some critical system, the cost incurred will be very high. The failures of noncritical systems are also expensive.

The cost of execution is greatly affected by the languages design. A language which require many runtime type checks prevents fast execution of the code irrespective of the quality of the compiler.

1.5 INFLUENCES ON LANGUAGE DESIGN

The basic design of programming languages are influenced by the following factors,

(1) Computer architecture.

(2) Programming methodologies.

1.5.1 Computer Architecture

The language design can be greatly influenced by the basic computer architecture. Von-Neumann computer architecture is the most commonly used architecture. The languages which are designed according to Von-Neumann architecture are called imperative languages.

In Von-Neumann architecture, same memory area is shared between both data and programs. The CPU whose function is to execute instructions, is separated from the memory. Hence, instructions and data must be transmitted from memory to CPU and the result obtained after computation must be sent back to memory. All computers since 1940 are based on Von-Neumann architecture. The layout of a Von-Neumann computer is shown in the Fig. 1.5.1.

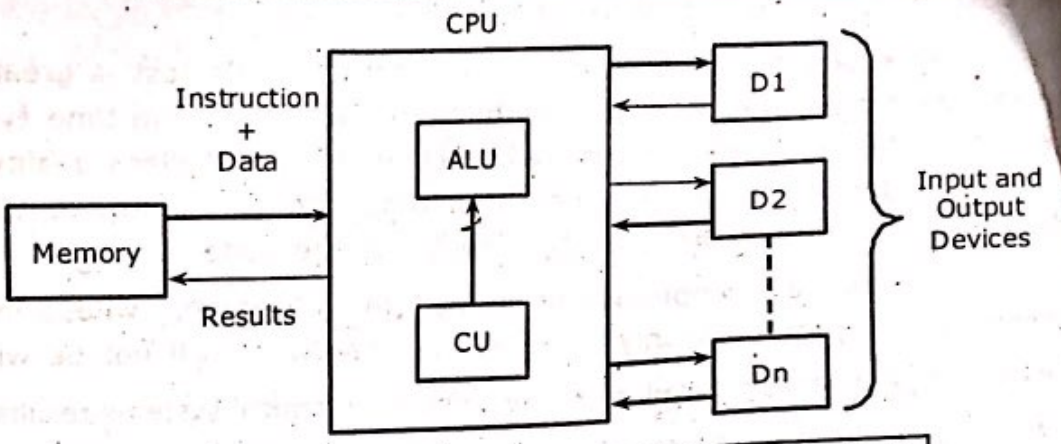


Fig. 1.5.1 The Von-Neumann Computer Architecture

The Von-Neumann architecture makes the imperative languages as variable which can model the, memory cells, assignment statements, based on the piping operation. The most efficient method of iterative form of repetition can also be modeled. Expression operands are transmitted from memory to the CPU and the result is sent back to the memory cell expressed at the left side of the assignment statement.

Advantage : The process of iteration is fast on this architecture as the instructions are stored in memory cells hence, eliminating the use of recursion.

In Von-Neuman architecture the execution of a machine code program occurs in a process called the fetch-execute cycle. Programs reside in memory but are executed in the CPU. Each instruction to be executed must be moved from memory to the processor. The address of the next instruction to be executed is maintained in a register called the program counter.

The below algorithm simply describes the fetch-execute cycle,

```
Initialize the program counter
repeat forever
    fetch instruction pointed to by the program counter
    increment the program counter to point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

In algorithm, "decode the instruction" step means the instruction is examined to determine what action it specifies. Program execution terminates when a stop instruction is encountered, although an actual computer a stop instruction is rarely executed. When the user program execution is complete, control transfers from the operating system to a user program for its execution and then back to the operating system.

1.5.2 Programming Methodologies

In late 1960s to early 1970s, several programming methods came into existence in order to overcome the deficiencies and to enhance the features of programming languages.

In the early 1980s, computers were used to solve simple problems such as solving a set of equations to simulate satellite tracks. Later, in the late 1960s and early 1970s, a detailed analysis on structured programming related to both software development process and programming language design were conducted.

This was done because of the increase in programming costs and decrease in hardware costs. Further, more difficult problems, such as providing-worldwide airline reservation systems and controlling petroleum refining facilities were solved by the computers.

During the analysis of 1970s, new programming methodologies called top-down design and step wise refinement were developed. Several deficiencies such as incompleteness of type checking and inadequacy of control statements were identified.

Later, in the late 1970's, data-oriented programming methodology came into existence. These methods focused mainly on the usage of data abstraction and data design. A language that is data-oriented must support the concept of data-abstraction. SIMULA 67 was the first language which gave support to data abstraction till its development, the advantages of data abstraction were not identified.

In the early 1980s, object-oriented methodology came into existence which is the extension of data-oriented methodology. A language that is object-oriented must support three features. They are,

(1) **Data Abstraction :** It encapsulates data objects and controls access to them.

(2) **Inheritance :** In order to reuse the existing software, inheritance is an effective feature. It is responsible for increasing the productivity in software development process. Object-oriented programming was supported by certain languages like, Small talk. Because of the limited use of Small talk, other imperative programming languages like Ada 95, C++ and Java and functional programming language, named CLOS were developed to provide support for object-oriented programming.

In procedure-oriented programming, an effective analysis has carried out in the area of concurrency. This leads to the need for language facilities for constructing and controlling concurrent program units. These capabilities are supported by some languages namely Ada and Java.

3) **Dynamic Method Binding :** Dynamic binding is the binding which occurs at run-time. It is used to build dynamic type hierarchies and to form 'abstract datatypes'. It is also used to provide more functional and efficient implementation to be selected out at run-time of the derived classes.

1.6 LANGUAGE CATEGORIES

Programming languages are categorized into four disciplines namely, imperative, functional, logic and object-oriented. In the previous sections the characteristics of imperative and functional programming languages are covered. Many object-oriented languages support the features of imperative languages such as expressions, assignment statements and control structures.

Example : Java control structures are similar to the control structures of C.

A logic programming language is a rule-based language. In an imperative language, an algorithm is specified and the order of execution of the statements can be predicted. In rule-based language, rules are not specified in a particular order and the implementation system should choose an execution order that produces the desired result. PROLOG is popular logic programming language.

Markup languages such as HTML are often confused with programming languages. However, markup languages are not used for computations but for displaying the content of a web page. In the recent past few scripting languages came into existence, which include the features of programming languages along with specific instructions required for web pages. JAVA scripts and VB script are of this type.

1.6.1 Imperative (or) Procedure-Oriented Programming

Imperative or procedural languages are command-driven or statement-oriented languages that are directly based on Von-Neumann architecture. Hence, the programmers of these languages must manage the variables and assign the values to them. The imperative family of languages begins with FORTRAN, PASCAL, C, C++, ALGOL, COBOL, ADA, Small talk etc. These are general purpose imperative languages.

These languages are fundamentally dependent on variables which holds values and these values are assigned to the variables by using an explicit assignment operator. The variables, at any instant, describes the state of computation.

Syntax

```
Statement 1;
Statement 2;
```

Advantages

(1) The efficiency of these languages is more than the functional languages.

(2) The state of computation of these languages can be determined at any instant of time by using the variables.

Disadvantages

(1) Imperative languages have complex syntactic and semantic structures.

(2) Increased labor is required for constructing the programs using imperative languages.

(3) Concurrent program execution is difficult to understand, design and use.

(4) Additional tasks such as maintaining cooperation among multiple concurrent tasks and static division of the programs is the sole responsibility of the programmer.

1.6.2 Object-Oriented Programming

In object-oriented programming, everything is considered as objects. In this case complex data objects are created and set of functions that can operate on those created objects are designed. Complex objects can inherit properties of the simpler objects. By creating concrete data objects, the object-oriented programs can gain efficiency over imperative languages.

Example

C++, Java, Ada and Smalltalk.

Advantages

(1) Complexity can be easily managed by using objects.

(2) Security of data is more when compared to procedural and functional oriented languages.

1.6.2.1 Features of Object-Oriented Programming

(1) Object-oriented programming is a programming methodology which consists of a set of interacting objects. These objects are organized in such a way that they can communicate with the other objects by sending and receiving messages.

(2) Object-oriented programming supports four fundamental concepts i.e., data abstraction, inheritance, encapsulation and polymorphism.

(3) Object-oriented programming languages support the paradigm with classes, methods, objects and message passing.

(4) In object-oriented programming everything looks like an object. An object is a group of procedures that share a state since, data is also part of a state, data and all the procedures that can be applied to it can be captured in a single object.

1.6.2.1.1 Object-Oriented Features Supported by C++

C++ is an object-oriented programming language that supports the following object-oriented features,

(1) **Classes :** A class is a collection of objects that share some common properties and relationships. It can hold both data and functions.

It is a user defined datatype that contains variables called the 'data members' and functions called the 'member functions'. The variables and functions together are called the members of a class. The member functions can operate on the data members of a class.

(2) **Objects :** Objects are the instances of a class. They are run-time entities that can store data as well as, send and receive messages. They occupy some space in memory and have the associated addresses.

(3) **Abstraction :** It is a technique of hiding the background (implementation) details thereby revealing only the necessary or relevant features of a program. In C++, it is used in classes where the attributes hold information and are operated by the functions.

It prevents the programmer from knowing the internal details. A programmer has to know which task is performed by which function without knowing anything about how that task is performed by the function.

(4) **Encapsulation :** It is a technique in object-oriented programming in which both the data and methods are wrapped (encapsulated/bind) together at one place so that any outside interference and misuses are prevented. It is an important feature of a class.

In encapsulation, the data is not accessible to the external world. Hence, only those methods or functions which are grouped in a class can access them.

(5) **Inheritance :** It is an object-oriented programming technique used to derive one class from the other in such a way that many additional features are added to the derived class apart from having the base class features.

Through inheritance, code reusability can be achieved. Reusability is the process of reusing the features of an existing class and implementing those features in a newly derived class.

Inheritance in C++ can exist in one of the following forms,

(i) **Single Inheritance :** The process of inheriting a derived class from a single base class is called single inheritance.

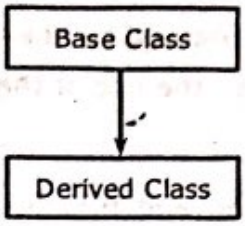


Fig. 1.6.1 Single Inheritance

(ii) **Multiple Inheritance :** The process of inheriting a derived class from more than one base classes is called multiple inheritance.

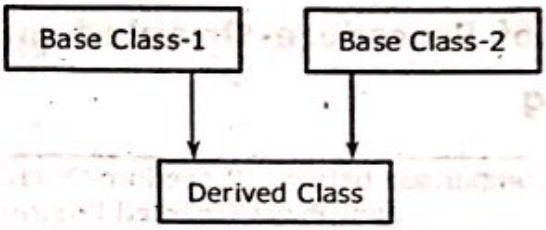


Fig. 1.6.2 Multiple Inheritance

(iii) **Hierarchical Inheritance :** The process of deriving many classes from a single base class is called hierarchical inheritance.

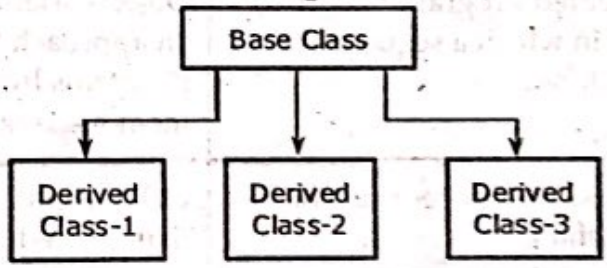


Fig. 1.6.3 Hierarchical Inheritance

(iv) **Multilevel Inheritance :** The process of deriving a class from another derived class is called multilevel inheritance.

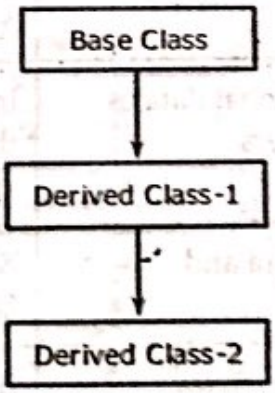


Fig. 1.6.4 Multilevel Inheritance

(v) **Hybrid Inheritance :** It is a combination of multilevel and multiple inheritance.

Hybrid inheritance = Multilevel inheritance + Multiple inheritance

(6) **Polymorphism :** It is an object-oriented technique that refers to the ability of a function to exhibit multiple meanings i.e., the use of the same function name to mean different things.

Example : Operator overloading and function-overloading.

(7) **Dynamic Binding :** It is a type of polymorphism in which the most appropriate function is called by the compiler at the run-time based on the number and type of arguments. It is also known as 'run-time polymorphism' or 'dynamic-linking'.

1.6.3 Comparison of Procedure-Oriented and Object-Oriented Programming

Table 1.6.1 Comparison between Procedure-Oriented Programming and Object-Oriented Programming

S.No.	Procedure-Oriented Programming (POP)	Object-Oriented Programming (OOP)
(1)	Procedure-Oriented Programming (POP) is an approach in which a sequence of tasks are to be done.	Object-Oriented Programming (OOP) is an approach that modularizes the programs by creating partitioned memory area for data and methods.
(2)	POP emphasizes on a procedure which involves doing things.	OOP emphasizes on data object rather than procedures.
(3)	In POP, large programs are divided into functions.	In OOP, large programs are divided into objects.
(4)	Data moves openly from one function to another function in POP.	Data is hidden and is difficult for external function to access in OOP.
(5)	In POP, functions share global data as there are no access specifiers.	In OOP, functions are tied together in the data structure because of access specifiers like private or public.
(6)	It is not possible to add data and functions.	New data and function can be easily added, when required.
(7)	It follows top-down approach for designing programs.	It follows bottom-up approach in designing.
(8)	Examples of POP languages are C, COBOL, FORTRAN.	Examples of OOP languages are C++, Java.

1.6.4 Functional (or) Applicative Programming

A functional or applicative language is one in which the primary means of making computation is by applying functions to given parameters. Programming in functional languages can be done without using any kind of variables, assignment statements and iterations. Thus, the functional languages, instead of focusing on the changes in state, emphasizes on the function that must be applied to the initial machine state.

In these languages, the programmers need not be concerned with the variables because the memory cells are not abstracted into the language. The development of a program proceeds by generating the functions from previously created functions to obtain more complex functions that can manipulate the initial data to obtain the final result.

Syntax

function_n(.....function_2(function_1 (data)).....)

Example : LISP(List Processor) and ML(Meta Language).

Advantages

(1) Functional languages have simple syntactic and semantic structures.

(2) Concurrent execution of programs is easy, as the programs are first converted into graphs which in turn can be executed through graph reduction methods.

(3) Less labour is required for constructing the programs.

(4) Cooperation synchronization is not the burden of a programmer.

Disadvantage : Less efficient than the imperative languages.

1.6.5 Logic (or) Rule-Based Programming

In rule-based languages, no specific order is followed in specifying rules. Thus, the language implementation system must select some order of execution that leads to the appropriate result. These languages can execute an action based on the satisfaction of certain enabling conditions. PROLOG is the most common rule-based (logic) programming language which consists of a class of predicate logic expressions as its enabling conditions. Execution of a rule based language is similar to an imperative language except that the statements are not sequential. Enabling conditions determine the order of execution.

Syntax

enabling condition_1 → Action_1

enabling condition_2 → Action_2

enabling condition_3 → Action_3

.

.

.

enabling condition_n → Action_n

1.6.6 Special Purpose and General Purpose Languages

The languages designed for a specific problem domain are called special purpose languages.

Example : PROLOG and GPSS.

General purpose languages are the languages designed for a wide range of problems, as they have wide range of applicability.

(or)

A type of programming language that is capable of creating various types of programs.

Example : COBOL, LISP, PERL and JAVA.

C is the best example of a general-purpose language developed by Dennis Ritchie in 1972.

Table 1.6.2 Differences between Special Purpose and General Purpose Languages

S.No.	Special Purpose Languages	General Purpose Languages
(1)	The languages designed for a specific problem domain are called special purpose languages.	The languages designed for wide range of problems are called general purpose languages.
(2)	They have narrow-range applicability.	They have wide-range applicability.
(3)	Example of such a language is PROLOG, which was designed to serve the narrow range of natural language processing	COBOL is a general-purpose language, which was designed to support all business oriented applications.
(4)	Other examples includes GPSS, which is used for systems simulation and RPG to produce business reports.	Other examples includes LISP, which is used for artificial intelligence and PERL and JAVA are used for Web-centre programming.

Examples

(1) **Reliability Vs Cost of Execution :** The increase in reliability resulting from index range checking in offset by an increase in the running time of a program.

Example : Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs.

(2) **Writability Vs Readability :** A language that allows you to express computations/ algorithms very briefly and also tends to be difficult to read.

Example : APL provides many powerful operators, allowing computations to be written in a compact program but at the cost of poor readability.

(3) **Writability Vs Reliability :** A language that gives the programmer the convenience/ flexibility of "breaking an abstraction" is more likely to lead to subtle bugs in a program.

Example : C++ pointers are powerful and very flexible but are unreliable.

1.8 IMPLEMENTATION METHODS

A computer has two primary components. They are,

(1) Internal memory and

(2) A processor

The internal memory stores programs and data whereas, the processor is a large combination of circuits that helps in realizing a set of primitive operations or machine instructions. In computers, the macroinstructions are implemented as a set of lower-level instructions called microinstructions. The machine language usually consists of a set of macroinstructions which are easily understandable by most of the hardware components in the absence of supporting software. The machine-level language provides most of the commonly used primitive operations, and need a system software for creating program interfaces in high-level languages.

Apart from language implementation system, a large collection of programs usually, called as 'operating system' is also required that supports high-level primitive operations such as resource management, file management, etc. A language implementation system is interfaced to an operating system but, not to a processor as it provides most of the facilities to the implementation system. The operating system and the language implementation form the layers of the machine language interface of a computer. These layers acts as virtual computer that provides interfaces to the higher-level users.

The layered view of a computer is shown in Fig. 1.8.1.

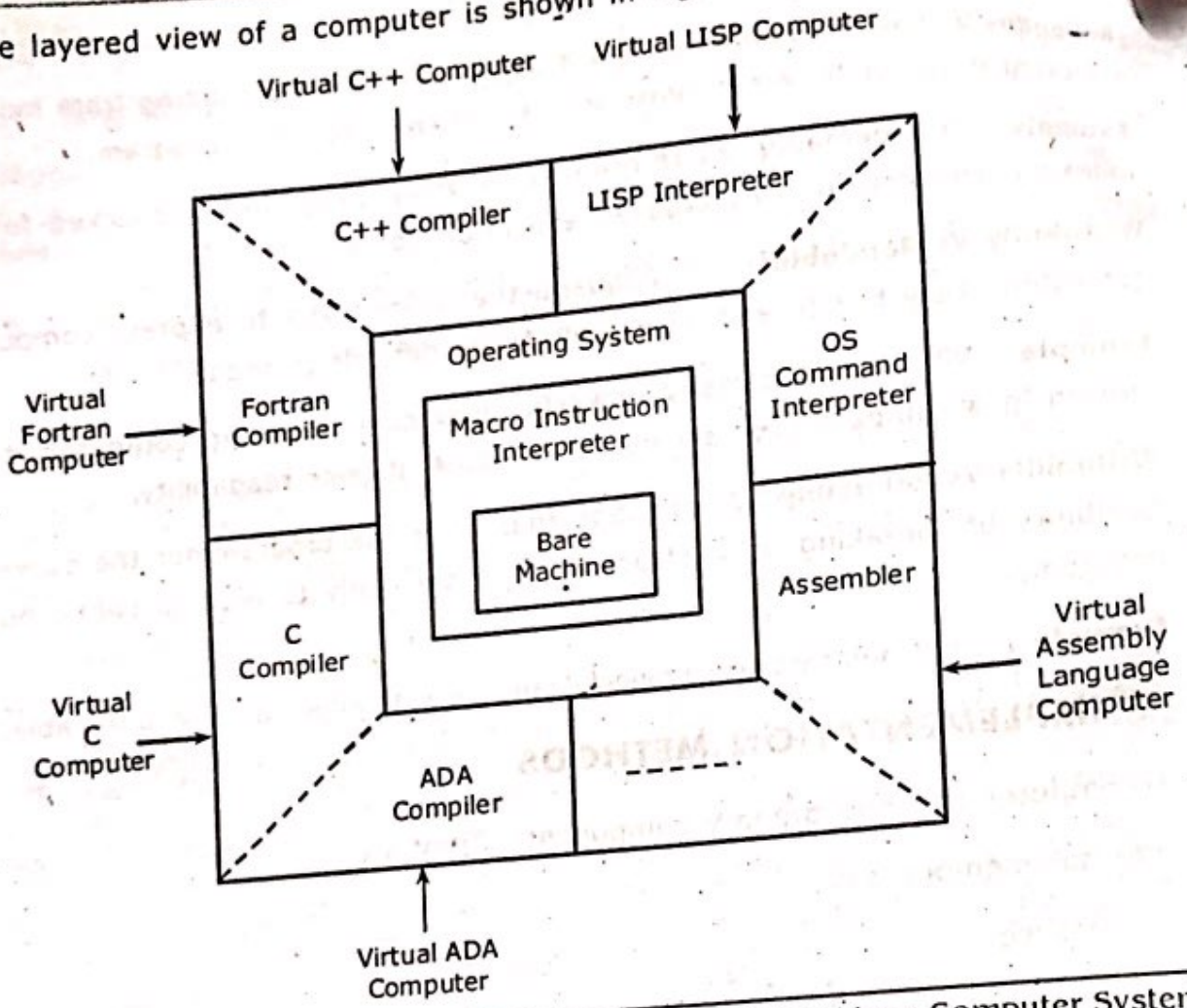


Fig. 1.8.1 Layered Interfaces of Virtual Computers Provided by a Computer System

Programming languages can be implemented by any one of the following methods,

(1) Compiler implementation.

(2) Pure interpretation.

(3) Hybrid implementation.

1.8.1 Compiler Implementation

In compiler implementation, programs written by the programmers are translate into machine language that can be executed directly on the computer. This implementatio is very fast. Most of the popular languages such as C, FORTRAN, COBOL and ADA us compiler implementation.

The phases involved in the process of compilation are shown in Fig. 1.8.2.

(1) ***Source Program*** : The program that is translated by a compiler is called a 'sour program'.

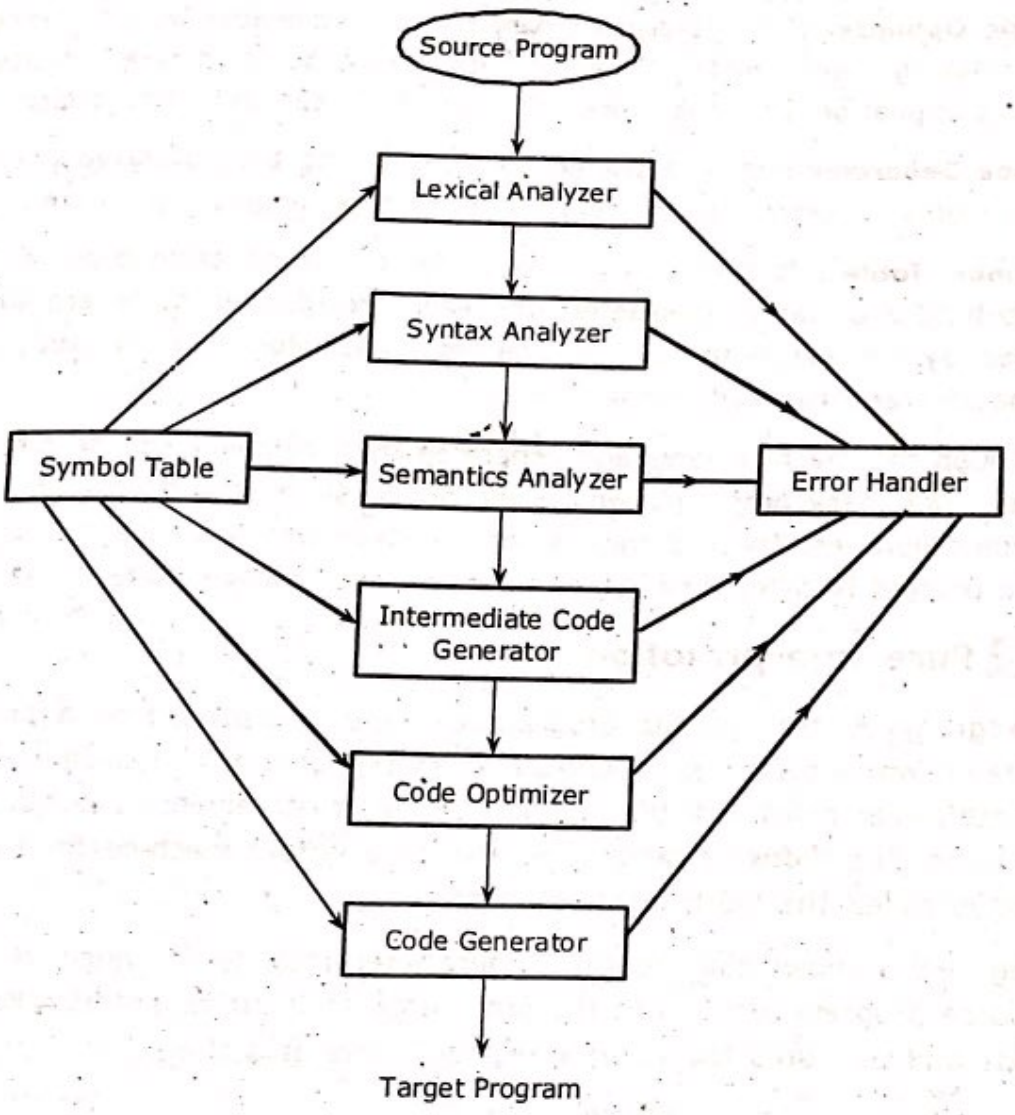


Fig. 1.8.2 The Compilation Process

xical Analyzer : The lexical analyzer accumulates the characters to form lexical units ich are identifiers, operators, special words and punctuation symbols. The nments in a source program are ignored by a lexical analyzer.

tax Analyzer : The syntax analyzer uses the lexical units, generated by a lexical lyzer and constructs the hierarchical structures called 'parse trees' which esents the syntactic structure of a program.

antic Analyzer : It is an integral part of the intermediate code generator which ks the validity and meaning of the parse tree i.e., it checks for the errors that difficult to find during syntax analysis such as type errors.

mediate Code Generator : At an intermediate stage between the source program he machine language program, a program in a different language is generated intermediate code generator.

(6) **Code Optimizer :** It is a discretionary part of compilation which improves programs by making them smaller, faster or both. This type of compiler is often used when the compilation speed is more important than the execution speed.

(7) **Code Generator :** It is responsible for translating the optimized intermediate code form of a program into its equivalent machine language program.

(8) **Symbol Table :** It acts as a database for the compilation process. The type and attribute information associated with each user-defined name are stored in symbol table by the lexical and syntax analyzers. This information is used by the semantic analyzer and the code generator.

Though the machine language generated by a compiler can be directly run on the hardware, but many programs require the previously executed programs that are stored in standard libraries. These programs can be linked with the user programs by the linker and the process is called 'linking'. After linking, the machine code is ready for execution.

1.8.2 Pure Interpretation

Programs written by the programmers are interpreted by a program called an 'interpreter' which serves as a software simulation of a machine. The fetch-execute cycle of an interpreter deals with the high-level program statements but not with the machine instructions. The software simulation acts as a virtual machine for the language. This method is called the 'pure interpretation'.

Fig. 1.8.3 shows the process of pure interpretation in which an interpreter takes the source program along with the other data as input to produce the desired output. It reads and executes the input statements one at a time.

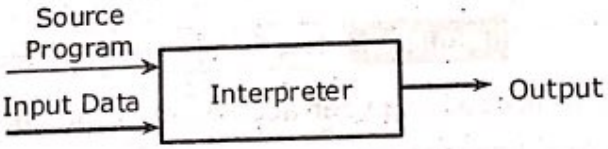


Fig. 1.8.3 Pure Interpretation

Comparison between Compilation and Interpretation

(1) **Interpretation can be More Flexible than Compilation :** Interpretation has great flexibility and better error detection capability than compilation. An interpreter directly runs the source program making it possible to make changes in a program thereby adding new features with more error correction ability.

In a compiler, all the translation is completed before running the target program which prevents it from being readily adapted as it runs.

(2) **Compilation can be More Efficient than Interpretation :** The performance of compilation is better than the interpretation. A decision made at the compile time can't necessarily made at the run-time.

Example : If the compiler can guarantee that the variable V, will always lie at a location 2006, it can generate machine language instructions that can access this location, whenever the source program refers to 'a'. On a contrary, an interpreter may need to look up in a table every time, when V is accessed in order to determine its location. Since the program is compiled only once but can be executed many times, the saving in time can be substantial.

The disadvantage of interpretation is that it requires more space. This is due to the fact that it requires source program along with the symbol table for interpretation. The source program must be stored in such a way that it can provide easy access and modification.

The disadvantage of compilation is that it requires machine time to compile a source program into target code.

1.8.3 Hybrid Implementation

Some language implementation systems translates, the programs written in high-level language to an intermediate language for easy interpretation and performs faster than the pure interpretation. Such systems are called 'hybrid implementation systems'.

The process of hybrid implementation is shown in Fig. 1.8.4 in which it interprets the intermediate code rather than translating the intermediate code to machine code.

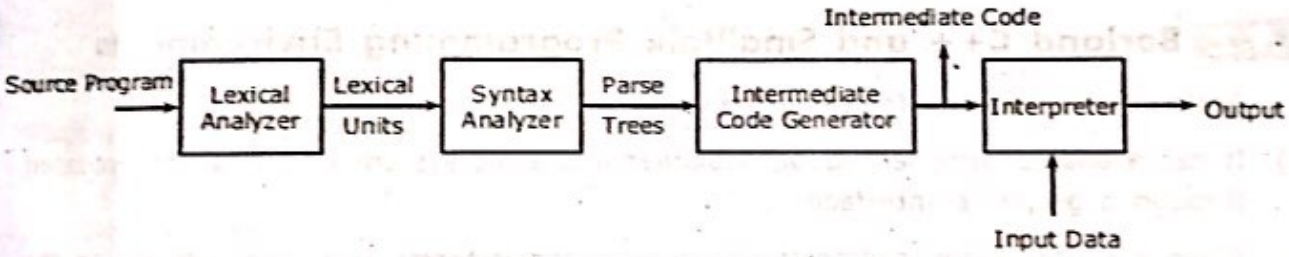


Fig. 1.8.4 Hybrid Implementation

The language 'perl' is implemented with a hybrid system. The Java in initial implementation is in hybrid form, with its byte code providing portability to any machine that has a byte-code interpreter and a run-time system (called the Java Virtual Machine).

An implementor may sometimes offers both compiled and interpreted implementation for a language. In such situations, an interpreter generates and debugs the program. Finally, they are compiled to increase their execution speed.

1.9 PROGRAMMING ENVIRONMENTS

Programming Environment refers to the collection of tools for developing software. The collection may include only a file system, a linker, a text editor and a compiler or it may comprise of a group of integrated tools which can be accessed through a uniform user interface.

To study programming environment, the following features must be considered.

(1) **Platform Independence :** Is the code designed for one platform to be used for other platforms?

(2) **User Distribution :** Is remote user access supported by the environment?

(3) **Distributed Processing :** Is distributed processing supported by the environment?

(4) **Data Distribution :** Is data located at different network locations be used easily?

(5) **Code Distribution :** Is code held on different sites be used transparently?

(6) **Extensibility :** How easy is to expand/enhance the capabilities of an environment for a specific application?

(7) **Efficiency :** How much efficient is the code generated by the environment?

(8) **Security :** How much security is provided by an environment?

1.9.1 Unix

Unix is an older operating system and tool collection. In 1970's it is firstly distributed around a portable multiprogramming operating system. It provides a powerful support tools for software production and maintenance in a variety of languages. Now-a-days often used through a GUI.

Example : Common Desktop Environment (CDE), (GNOME and KDE) that runs on top of UNIX.

1.9.2 Borland C++ and Smalltalk Programming Environments

Borland C++ has the following features,

(1) It has a unified compiler, editor, debugger and a file system which can be accessed through a graphical interface.

(2) When a syntax error is detected by a compiler, it terminates and switches to the editor and leaves the cursor at the point of error in the source program.

Smalltalk is an integrated programming environment and a more elaborative complex and power language than Borland C++,

(1) It is a system that uses a windowing system for the first time.

(2) To provide a uniform interface to all the available tools it makes use of a mouse pointing device.

1.9.3 Microsoft Visual C++ Programming Environment

(1) Microsoft Visual C++ consists of an elaborative collection of software development tools. A windowed interface is provided for using these tools.

(2) This is used in conjunction with other similar systems such as Visual Basic, Delphi and Java Development Kit to provide easier ways for constructing GUIs to programs.

1.9.4 C Programming Environment

Advantages

(1) C language enables the user to define macros using a special program called C processor.

(2) Macro's in C reduces the code and can be lined directly into the code, thereby reducing the overhead of function call.

(3) C provides a rich set of library functions so as to support different input/output arithmetic operations string handling functions.

(4) C contains fewer syntactical rules which makes C compiler highly optimized (i.e., it will occupy only 256 k of total memory).

(5) C allows the data to be viewed in many ways since it provides limited type checking.

(6) C program executes at a speed which is almost equivalent to the assembly language programs.

(7) C supports modular programming i.e., it supports separate compilation and linking.

(8) C pre-processor allows the users to include other source files within its source file.

(9) C provides a rich set of bit manipulation operators and can perform pointer arithmetic.

(10) C include the concept of pointers so as to enhance the speed of a program.

(11) C consists of entire of set of control structures with a very flexible semantics.

(12) C programs are highly cross-platform portable and very memory efficient.

Disadvantages

(1) C doesn't support data abstraction and generic abstraction due to which it is not possible to perform modularity and reusability.

(2) It is highly unreadable as it doesn't support runtime checking due to which many errors (bugs) remain undetected.

(3) It is easy to misuse the low-level features of C language due to which, it is very difficult to maintain the code.

1.10 EVOLUTION OF THE MAJOR PROGRAMMING LANGUAGES

Programming languages fall into two fundamental categories - low and high-level languages. Low-level languages are machine-dependent; that is, they are designed to be run on a particular computer. In contrast, high-level languages (for e.g., COBOL and BASIC) are machine-independent and can be run on a variety of computers.

Through the first four decades of computing, programming languages evolved in generations. The first two generations were low-level and the next two high-level generations of programming languages.

The higher-level languages do not provide us a greater programming capabilities, but they do provide a more sophisticated programmer/computer interaction. In short, the higher the level of the language, the easier it is to understand and use.

Example : In a fourth-generation language you need only instruct the computer system what to do, not necessarily how to do it.

When programming in one of the first three generations of languages, you have to tell the computer what to do and how to do it. What comprises a new generation is less clear, therefore, languages after the fourth-generation are referred to as a very high level languages.

NOTE : ***Low level languages are machine dependent while high level languages are machine independent.***

1.10.1 Evolution of Low-Level Languages

All computers operate by following machine language programs, a long sequence of instructions called machine code that is addressed to the hardware of the computer and is written in binary notation, which uses only the digits 1 and 0. First-generation languages, called machine languages, required the writing of long strings of binary numbers to represent such operations as "add", "subtract" "and compare".

Because writing programs in machine language is impractical (it is tedious and error prone), symbolic or assembly, languages - second-generation languages-were introduced in the early 1950s. They use simple mnemonics such as A for "add" or M for "multiply", which are translated into machine language by a computer program called an assembler. The assembler then turns that program into a machine language program. An extension of such a language is the macro instruction, mnemonic (such as "READ") for which the assembler substitutes a series of simpler mnemonics. The resulting machine language programs, however, are specific to one type of computer and will usually not run a computer with a different type of Central Processing Unit (CPU).

1.10.2 Evolution of High-Level Languages

(1) Early 50's

(i) Early HLL's started to emerge.

(ii) FORTRAN

(a) Stands for FORmula TRANslating system.

(b) Developed by a team led by John Backus at IBM for the IBM 704 machine.

(c) Successful in part because of support by IBM.

(d) Designed for scientific applications.

(e) The root of the imperative language tree.

(f) Lacked many features that you now take for granted in programming languages,

- Conditional loops.
- Statement blocks.
- Recursive abilities.

(g) Many of these features were added in future versions of FORTRAN.

- FORTRAN II, FORTRAN IV, FORTRAN 77, FORTRAN 90.

(h) Had some interesting features that are now obsolete.

- COMMON, EQUIVALENCE, GOTO.
- We may discuss what these are later.

(2) Late 50's

(i) COBOL

(a) COmmon Business Oriented Language.

(b) Developed by US DoD.

(c) Separated data and procedure divisions. But didn't allow functions or parameters.

(d) Still widely used, due in part to the large cost of rewriting software from scratch. Big companies would rather maintain COBOL programs than rewrite them in a different language.

(ii) LISP

(a) LISt Processing.

(b) Developed by John McCarthy of MIT.

(c) Functional language.

(d) Good for symbolic manipulation, list processing.

(e) Had recursion and conditional expression. Not in original FORTRAN.

(f) At one time used extensively for AI.

(g) Today most widely used version, COMMON LISP, has included some imperative features.

C

(a) Developed by Dennis Ritchie to help implement the Unix Operating System.

(b) Has a great deal of flexibility, specially with types.

- Incomplete type checking.
- Void pointers.
- Coerces many types.

Prolog

(a) Logic programming.

(b) Still used somewhat, mostly in AI.

s

Ada

(a) Developed over a number of years by DoD.

(b) Goal was to have one language for all DoD applications.

- Especially for embedded systems.

(c) Contains some important features.

- Data encapsulation with packages.
- Generic packages and subprograms.
- Exception handling.
- Tasks for concurrent execution.

(d) Very large language - difficult to program reliably, even though reliability was one of its goals!

(e) Early compilers were slow and error-prone.

(f) Did not have the widespread general use that was hoped.

(g) Eventually the government stopped requiring it for DoD applications.

- Use faded after this.
- Not used widely anymore.

(h) Ada 95 added object-oriented features.

- Still wasn't used much, especially with the advent of Java and other OO languages.

(ii) Smalltalk

(a) Designed and developed by Alan Kay.

- Concepts developed in 60's but language did not come to fruition until 1980.
- Designed to be used on a desktop computer 15 years before desktop computers existed.

(b) First true object-oriented language.

(c) Language syntax is geared toward objects.

- "messages" passed between objects.
- "methods" are involved as responses to messages.

(d) Always dynamically bound.

- All classes are subclasses of object.

(e) Also included software level environment.

(f) Had large impact on future OOLs, esp. Java.

(iii) C++

(a) Developed largely by Bjarne Stroustrup as an extension to C.

- Backward compatible.

(b) Added object-oriented features and some additional typing features to improve C.

(c) Very powerful and very flexible language.

(d) But still has reliability problems.

- Ex. no array bounds checking.
- Ex. dynamic memory allocation.

(e) Widely used and likely to be used for a while longer.

(iv) Perl

(a) Developed by Larry Wall.

(b) Takes features from C as well as scripting languages awk, sed and sh.

(c) Some features,

(iii) ALGOL

(a) ALGOL 58 and then ALGOL 60 both designed by international committee.

(b) Goals for the language.

- Syntax should be similar to mathematical notation and readable.
- Should be usable for algorithms in publications.
- Should be compilable into machine code.

(c) Included some interesting features.

- Pass by value and pass by name parameters.
- Recursion (first in an imperative language).
- Dynamic arrays.
- Block structure and local variables.

(d) Introduced Backus-Naur From (BNF) as a way to describe the language syntax. Still commonly used today, but not well-accepted at the time.

(e) Never widely used, but influenced virtually all imperative languages after it.

(3) Late 60's

(i) Simula 67

(a) Designed for simulation applications.

(b) Introduced some interesting features.

- Classes for data abstraction.
- Coroutines for re-entrant subprograms.

(ii) ALGOL 68

(a) Emphasized orthogonality and user-defined data types.

(b) Not widely used.

(4) 70's

(i) Pascal

(a) Developed by Nicklaus Wirth.

(b) No major innovations, but due to its simplicity and emphasis of good programming style, became widely used for teaching.

- Regular expression handling.
- Associative arrays.
- Implicit data typing.

(d) Originally used for data extraction and report generation.

(e) Evolved into the archetypal Web scripting language.

(f) Has many proponents and detractors.

(g) **pd's**

(i) Java

(a) Interestingly enough, just like Ada, Java was originally developed to be used in embedded systems.

(b) Developed at Sun by a team headed by James Gosling.

(c) Syntax borrows heavily from C++.

- But many features of C++ have been eliminated.
- No explicit pointers or pointer arithmetic.
- Array bounds checking.
- Garbage collection to reclaim dynamic memory.

(d) Object model of Java actually more resembles that of Smalltalk rather than that of C++.

- All variables are references.
- Class hierarchy begins with object.
- Dynamic binding of method names to operations by default.

(e) But not as pure in its OO features as Smalltalk, due to its imperative control structures.

(f) Interpreted for portability and security.

- Also JIT compilation now.

(g) Growing in popularity, largely due to its use on Web pages.

(7) 00's

(i) C#

(a) Main roots in C++ and Java with some other influences as well.

(b) Used with MS .NET programming environment.

(c) Some improvements and some deprovements compared to Java.

(d) Likely to succeed given MS support.

(4) **Fourth Generation Languages (4GL) :** The fourth generation programming languages are very closer to human languages.

Example : ORACLE, VB, VC++, SQL etc. are called as 4GL languages.

Most of the 4GL languages are used to access the databases. They allow the programmer to define 'What' is required without telling the computer 'how' to implement it.

Features of 4GL Languages

(i) Easy to use.

(ii) Limited range of functions.

(iii) Availability of options.

(iv) Default options.

1.2 REASONS FOR STUDYING CONCEPTS OF PROGRAMMING LANGUAGES

Following are the list of potential benefits of studying concepts of programming languages,

(1) **To Increase the Capacity of Expressing Ideas :** Programmers in the process of developing software feels difficult to idealize structures either verbally or in writing. The language which programmers used to develop software imposes restrictions on some control structures, data structures and abstractions they use. Hence, the algorithms constructed by them are also restricted. These restrictions or limitations can be minimized by understanding the features of various programming languages and by learning new constructs in the programming languages.

The features of one language can be simulated in other languages, doing so, will not reduce the importance of designing languages. However, it is a best practice to use a feature whose design is integrated into a language rather than using a simulation of that feature (As it makes the language less elegant and unmanageable).

(2) **To Improve the Knowledge of Selecting an Appropriate Language :** Programmers with knowledge in few languages cannot judge a well suited programming language for the assigned project and hence they often use the language with which they are well known or familiar with, even if it is not appropriate for the new project.

If the programmer is familiar with the other available languages and particularly some features of these languages then, he can make a better choice in selecting an appropriate language.

(3) **To Increase the Learning Ability of a Programmer to Learn New Languages :** Now-a-days, programming languages are in a state of continuous evolution. Learning a new programming language is a lengthier and a difficult process, especially for those who are familiar with a few languages and who never assessed the concepts of programming language, in general.

After understanding the basic concepts of a programming language it would be easier to analyze how these concepts or features are integrated in the new language.

(4) **To Have a Better Understanding of the Importance of Implementation :** The implementation issues that affects a programming language must be considered while learning it. This, increases the ability of a programmer to intelligently use a language. Certain kinds of bugs encountered during program execution can be easily found by the programmer who is aware of the related implementation details.

It also allows us to conceptualize how various constructs of a programming language are executed by a computer.

Example : A programmer who is unaware of or knows less about the implementation of recursion doesn't know that a recursive algorithm is very much slower than its equivalent iterative version.

(5) **To Increase the Designing Ability of a Programmer for Designing New Languages :** Most professional programmers, sometimes design languages of one kind or the other.

Example : Most software systems require only small amount of data and commands to be entered by a user in order to obtain the desired output.

Designing a user interface is a complex design problem in some systems like word processor in which a user needs to traverse several levels of menus and enter a variety of commands.

Hence, a complete analysis of a programming language is needed in order to design a language for some complex systems, as it helps the users to assess and evaluate such products.

(6) **To Achieve an Overall Advancement in Computing :** The popularity of a programming language can be easily determined but the most popular languages are not always the best available languages.

Sometimes, a programming language is used widely because the programmers are not well aware of the concepts of programming languages in order to choose the best language for programming.

Consider the popularity of ALGOL-60, many people believe that it has better control statements than FORTRAN, hence it replaced FORTRAN, but the problem with ALGOL-60 is that it is very difficult to read and understand. In addition to the previously mentioned problem it doesn't specify the benefits associated with recursion, block structure and well-structured control statements, hence, the programmers failed to find the benefits of ALGOL-60 over FORTRAN.