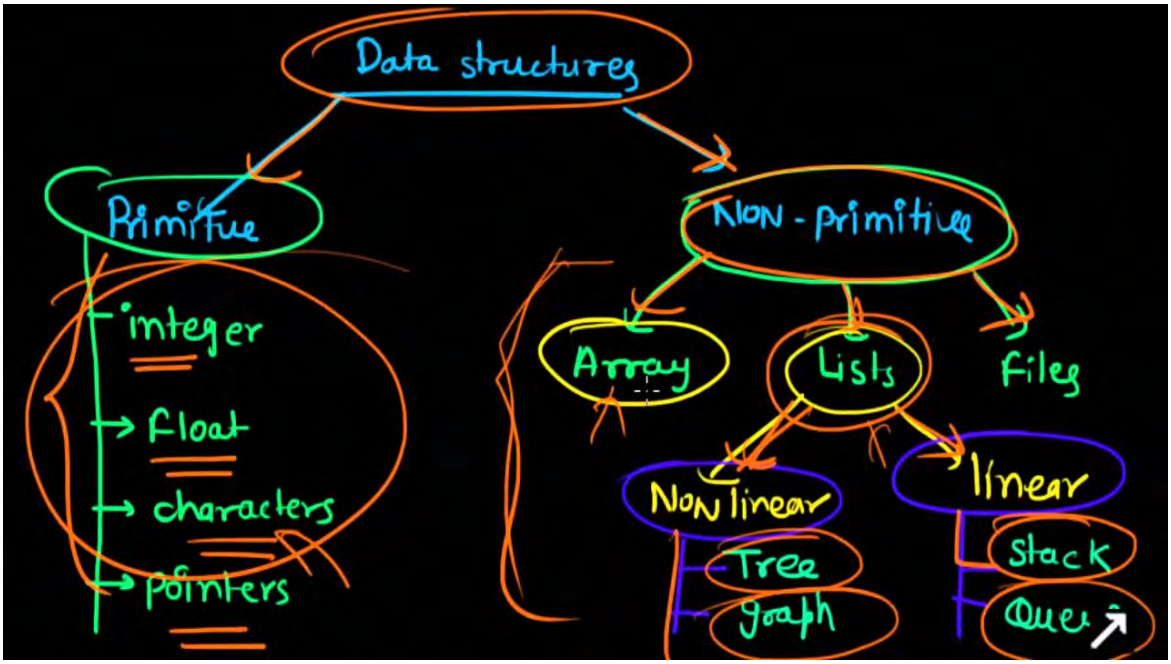


Programming Languages

Fundamental Concepts

Fundamental Concepts

PROGRAMMING LANGUAGE

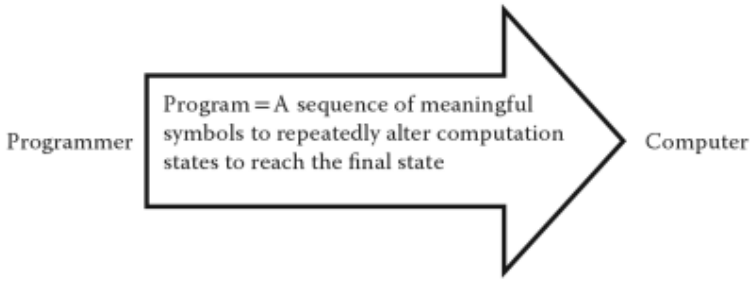
- A programming language is nothing but a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. The term programming language usually refers to high-level languages, such as BASIC, C, C++.
- High-level programming languages are more complex than the languages the computer actually understands, called machine languages.
- Lying between machine languages and high-level languages are languages called *assembly languages*. Assembly languages allow a programmer to substitute names instead of numbers as in machine languages.
- Lying above high-level languages are languages called *fourth-generation languages* (usually abbreviated 4GL), represents the class of computer languages closest to human languages.

PROGRAM AND COMPONENTS

When asked to automate a process or to solve a problem, a system analyst has to make a model of the system, parameterize the input and output behaviour, and connect various modules using a flow chart. These modules abstract the real-world process. The programs are specifications of the solutions that are developed to process data and handle dataflow between these modules.

The specification of a solution to a real-world problem is described at a *high level* for the ease of human understanding, and the computer's action is based on *low-level* machine instructions, there is a need for a translation process that uniquely maps (without any ambiguity) high-level instructions to a sequence of equivalent low-level instructions. Instructions should have a clear unique meaning to avoid ambiguity.

A program is a sequence of meaningful symbols to formally specify the solution to a complex problem (see Figure).



A program has three major components: logic + abstraction + control. Logic means coming up with the high-level specification of a solution to a problem. This requires repeated breaking up of a complex problem into a structured combination of simpler problems, and

combining the solutions of these simpler problems using well-defined operations to compute the final solution.

Abstraction means modelling an entity by the desired attributes needed to solve the problem at hand. The entity may have many more attributes. However, all the attributes may not be needed to model the solution of the problem. The advantage of abstraction is that programs arc easily comprehended and easily modified resulting into case of program maintenance.

Control means mapping the solution of the problem based on the von Neumann machine, where the memory of the computer is continuously altered to derive a final state of computation that contains a solution. Every instruction alters the state of computation to a new state. The use of explicit control in a program gives a programmer the power to modify the computer memory explicitly. The modification of computer memory to realize the logic varies from programmer to programmer, and makes a program difficult to understand.

Example

This example illustrates the three components through a simple, bubble-sort program that sorts a sequence of unsorted numbers. Bubble sort utilizes "repeatedly find the next maximum" strategy on progressively smaller subsequences. A program for bubble sort uses three components as follows:

- **Abstraction**: This represents the set of numbers as an indexible sequence.
- **Logic**: This compares adjacent numbers and swaps the position of numbers if the following number is smaller until the end of the sequence is reached. Comparisons of numbers in the current sequence gives the maximum of the current sequence. This element is excluded in future comparisons, and the comparison process is repeated with the remaining elements until one element is left.
- **Control**: This uses the code to repeatedly exchange and update the values in the different memory locations associated with variables.

A program is organized using multiple units: (1) program name, (2) imported routines from the software libraries or modules developed in the past, (3) declaration of the type information and variables needed for expressing the logic, (4) parameters to exchange the information between various program modules, and (5) sequence of commands to manipulate the declared variables.

TYPES OF PROGRAMMING LANGUAGES

We classify programming languages by their level.

- Natural language.
- Machine level language.
- High-level language.
- Assembly level language.
- Scripting languages.

Natural Languages

- A language spoken, written or signed by humans for general purpose communication.

Machine Level Language

- It is the lowest-level programming language.
- Machine languages are the only languages understood by computers.
- Programs written in high-level languages are translated into machine language by a compiler.
- A computer's machine language consists of strings of binary numbers (0,1).

Limitations

- In this programming language a programmer has to remember dozens of code numbers or commands in the machine instruction set.
- A programmer has to keep track of the storage locations of data and instructions.
- Modifications of a machine level program and the location of errors in it is a tedious job and can take long time.

Assembly Level Language

- The assembly language is simply a symbolic representation for its associated machine language.
- This representation consists of mnemonic operation codes and symbolic addresses.

Advantages

- Less number of errors and also errors are easy to find.
- Modification of assembly language programs is easier than that of machine language program.

Limitation

- They are machine oriented.

High Level Language (HLL)

A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages.

Characteristics of High Level Programming Language

- In computing, a high-level programming language is a programming language with strong abstraction from the details of the computer.
- In comparison to low-level programming languages, it may use natural language elements, be easier to use, or be more portable across platforms.
- Such languages hide the details of CPU operations such as memory access models and management of scope.

Scripting Language

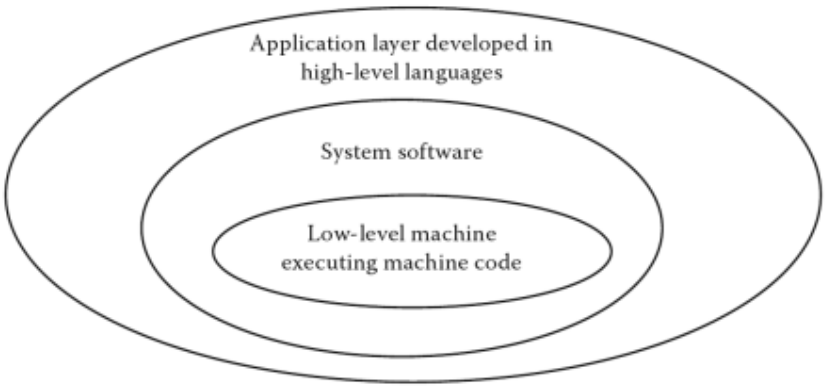
- A scripting language is a high-level programming language that is interpreted by another program at runtime rather than compiled by the computer's processor as other programming languages (such as C and C++) are.
- Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a web page, such as different menu styles or graphic displays or to serve dynamic advertisements.

EXECUTION OF PROGRAMS

Major criteria for the sound execution of a program are as follows:

1. The language constructs of a programming language should be well defined.
2. There should be a unique meaning for every language construct as computers do not handle ambiguities.
3. Each high-level instruction should be translated to a sequence of low-level instructions that perform consistently the same action on a computer every time.
4. A computer should execute consistently the same sequence of low-level instructions producing the same final result.

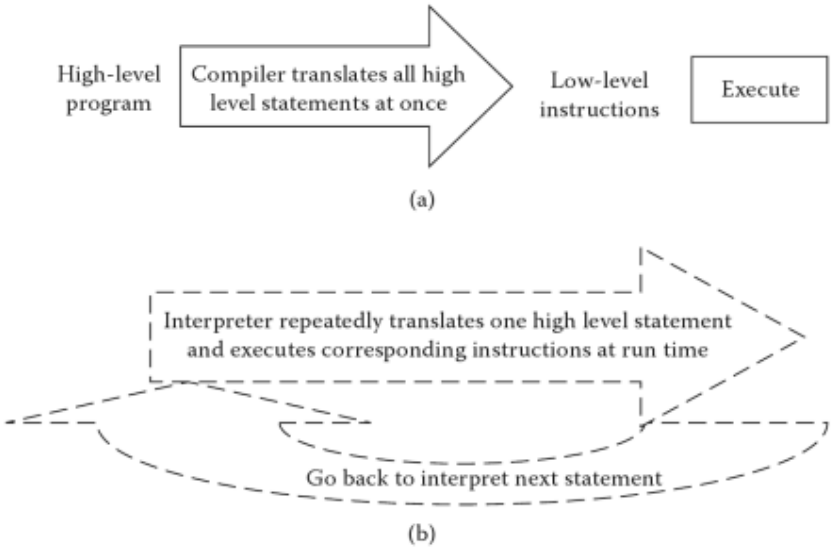
There are multiple layers of software before the programming language layer, as shown in following figure.



At the lowest level, it is the bare machine and machine code. The next level is the operating system, system utilities, and language translators. The outermost layer is the high-level programming and application layer. A high-level program is translated to low-level instructions and uses multiple intermediate layer interfaces to use the computer system resources and utilities needed for program execution.

There are three ways in which a high-level instruction can be translated to a set of low-level instructions: (1) compile the high-level instructions before the execution of the program; (2) interpret the high-level program, and execute it using an implemented abstract machine; and (3) just-in-time compilation of the high-level language that gives an effect of partially compiled code and partially interpreted code.

As illustrated in following figure, the process of compilation translates the high-level instructions to low-level instructions before execution.



Compilation versus interpretation, (a) A schematics of compilation (b) a schematics of interpretation.

On the other hand, interpreters translate and execute one instruction at a time. In contrast to compiled code, interpreted code goes through the translation and execution cycle for very statement.

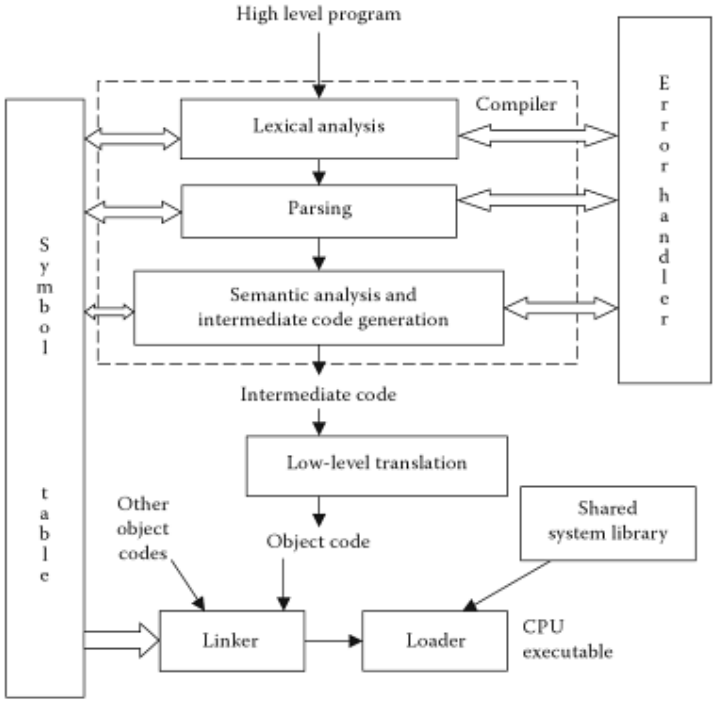
An advantage of compiled code is that it has no overhead of translation at run time. Another advantage of using compilers is that a large percentage of the programming bugs are detected, and memory allocations are optimized before execution.

The execution efficiency of interpreted code is an order of magnitude slower than the execution efficiency of the compiled code, because the translation process is interleaved with execution at run time. Another drawback of interpreters is that all the errors cannot be detected before program execution, and interpreted programs may crash after executing many instructions, making it unsafe for mission-critical programs. However, interpreters arc easy to

develop and have been used in early days lor the languages for which compiler technology was not well developed.

Modern-day compilers use a two-stage translation for execution ol a high-level program on a variety of architectures. Different architectures have different assembly languages, and it is not possible to write compiler for multiple architectures. Instead, an intermediate code has been developed for the programming languages. The first stage of compiler translates high-level programs to an intermediate-level code that is independent of computer architectures. The second stage translates intermediate code to low-level machine code.

The first stage of translation consists of (1) lexical analysis, (2) parsing, and (3) semantic analysis and code generation to generate intermediate code, as shown in following figure.



A lexical analyzer checks for the reserved words—meaningful words that are part of a language, identifiers, variables and numbers—and converts them into an internal representation called tokens for the case of parsing. The output of lexical analysis is a tokenized stream that becomes input to a parser. A parser validates the structure of a program sentence according to the grammar of a programming language. Its output is in the form of a tree for the ease of efficient internal handling during code generation. The semantic analyzer validates that parsed sentences are meaningful, and the code generator linearizes the tree-based representation to the corresponding intermediate code—a sequence of low-level abstract instructions. The optimization level removes the redundant code fragments and enhances the use of processor registers to improve the execution efficiency of the executable code. A symbol table is used to store the information from previous stages that may be needed or may need resolution in the following stages. The type of information that a symbol table contains arc variable names and their locations, nesting level of procedures, information

about nonlocal variables, where a procedure has been called, what procedure has been called, and so on.

In addition to two-stage compilation, a linker links multiple object files (compiled code) into one big executable code. Linking is done in the order specified by the programmer in the link command and is independent of the calling pattern of the procedures. The compiled code of a subprogram occurs only once in the executable code and is independent of the frequency of the calling of the subprogram. After linking, the information about the relative location of procedures is fixed. A loader loads the linked code to be executed in a memory segment—the memory area given to a user process corresponding to the executable code. The loader relocates the logical address derived after linking to the physical address in the RAM. The loaded executable program is called a process and is executed using a combination of operating system software and hardware techniques you will study in a course on operating systems. Dynamic link libraries arc shared system libraries to share the executable code by multiple processes. You will learn about the mechanisms of dynamic link libraries in a course on operating systems.

HIGH-LEVEL PROGRAMMING LANGUAGE TOOLS : COMPILER, LINKER, INTERPRETER

Compiler

- Compiler is used to transform a program written in a high-level programming language from source code into object code.
- Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program.
- The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code.
- The final step in producing an executable program — after the compiler has produced object code — is to pass the object code through a linker. The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.

Linker

- Also called link editor and binder, a linker is a program that combines object modules to form an executable program.
- Many programming languages allow you to write different pieces of code, called modules, separately.
- This simplifies the programming task because you can break a large program into small, more manageable pieces.
- Eventually, though, you need to put all the modules together. This is the job of the linker.

Interpreter

- An interpreter translates high-level instructions into an intermediate form, which it then executes.
- Compiled programs generally run faster than interpreted programs.
- The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long.
- The interpreter, on the other hand, can immediately execute high-level programs.

Editor

An editor is software, where the programmer can write the source code, edit it as well as compile and execute it. Like compilers and interpreters, the editors are also different for different programming languages.

POPULAR PROGRAMMING LANGUAGES

FORTRAN (FORmula TRANslation)

- The name FORTRAN is an acronym for FORmula TRANslation, because it was designed to allow easy translation of math formulas into code.
- Often referred to as a scientific language, FORTRAN was the first high-level language, using the first compiler ever developed. Prior to the development of FORTRAN computer programmers were required to program in machine/assembly code, which was an extremely difficult and time consuming task.
- The objective during it's design was to create a programming language that would be: simple to learn, suitable for a wide variety of applications, machine independent, and would allow complex mathematical expressions to be stated similarly to regular algebraic notation.

COBOL (Common Business Oriented Language)

- COBOL (pronounced /?ko?b?l/) is one of the oldest programming languages. Its name is an acronym for COmmon Business-Oriented Language, defining its primary domain in business, finance, and administrative systems for companies and governments.
- The COBOL 2002 standard includes support for object-oriented programming and other modern language features.
- COBOL was an effort to make a programming language that was like natural English, easy to write and easier to read the code after you'd written it.

BASIC (Beginner's All-purpose Symbolic Instruction Code)

- In computer programming, BASIC (an acronym for Beginner's All-purpose Symbolic Instruction Code) is a family of high-level programming languages.
- It is developed to provide computer access to non-science students.
- BASIC remains popular to this day in a handful of highly modified dialects and new languages influenced by BASIC such as Microsoft Visual Basic. As of 2006, 59 % of developers for the .NET platform used Visual Basic -NET as their only language.

PASCAL

- Pascal is an influential imperative and procedural programming language, designed in 1968/9 and published in 1970 by Nicklaus Wirth as a small and efficient language intended to encourage good programming practices using structured programming and data structuring.
- A derivative known as Object Pascal was designed for object oriented programming.

C Language

- C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system.
- Although C was designed for implementing system software, it is also widely used for developing portable application software.
- C is one of the most popular programming languages. It is widely used on many different software platforms.

C++ Language

- C++ (pronounced "C plus plus") is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language.
- It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
- It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C programming language and originally named "C with Classes". It was renamed to C++ in 1983.

JAVA

- Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun
- Microsystems' Java platform.
- The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.

- Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

LISP (LISt Processing)

- Lisp (or LISP) is a family of computer programming languages.
- It is the second-oldest high-level programming language in widespread use today.
- Lisp was originally created as a practical mathematical notation for computer programs.
- The name LISP derives from "LISt Processing". Linked lists are one of Lisp languages major data structures, and Lisp source code is itself made up of lists.

CHARACTERISTICS OF GOOD PROGRAMMING LANGUAGE

Clarity, Simplicity, and Unity

A programming language provides a frame work for thinking about algorithms. It should provide a clear, simple and unified set of concepts that can be used as primitives for developing an algorithm. To this end, it desirable to have a minimum number of different concept, with the rules for their combination being as simple and regular as possible. This attribute is called as conceptual integrity.

The syntax of a language affects the ease with which a program may be written, tested and later understood and modified. The programs should be readable. Semantic differences should be mirrored in the language syntax.

Orthogonality

The term orthogonality refers lo attributes of being able to combine various features of a language in all possible combinations, with each combination being meaningful.

For example, suppose a language provides an expression that can provide a value, it also provides a conditional statement that evaluates an expression lo get a true or false value. These two features of the language expression and conditional statements are orthogonal if any expression can be used within the conditional statement.

When the features of a languages are orthogonal the languages is easier to learn and programs are easier to write.

Naturalness for the Application

A language needs syntax to reflect the underlying logical structure of the algorithm. It should

be possible to translate such program statements that reflect the structure of the algorithm.

The language should provide appropriate data structures, operations, control structures and a natural syntax for the problem to be solved.

Support for Abstraction

"*Abstraction refers to the act of representing essential features without including background details or explanation*."

In C++, class supports data abstraction, so it is also called as abstract data type. Ideally the language should allow data structures, data types, and operations (fundamental macros) to be defined and maintained as self-contained abstractions. The programmers may use them in other parts of programs knowing only their abstract properties without concern for the details of their implementation.

Ease of Program Verification

The reliability of programs written in a language is always a central concern. There are many techniques for verifying that a program correctly performs its required function.

A program may be tested by executing it with test input data and checking the output results against the specifications. A language that makes program verification difficult is troublesome to use.

Simplicity of semantic and syntactic structure is a primary aspect that tends to simplicity of the programs.

Programming Environment

"*Programming environment is the environment in which programs are created and tested*."

The technical structure of programming language affects its utility. The presence of programming environment may nuke a weak language easier lu work with than a technically stronger language.

The availability of a reliable, efficient and well-documented implementation of the language must head the list. Special editors and testing packages speed the creation and testing of programs.

Programming environment consists of set of tools and a command language for invoking them. Typical tools are editors, debugger, verifiers, and test data generators.

Portability of the Programs

One important criterion for many programs is its transportability from the computer on which they are developed to other systems. A language that is widely available and whose definitions arc independent of the features of a particular machine forms a base for the production of transportable programs.

FORTRAN. C and Pascal all have standard definitions allowing for portable applications to be implemented.

Cost

The ultimate total cost of programming language is a function of many of its characteristics as given below:-

- **Cost of Training.** There is a cost of training programmers to use the language. This is a function of simplicity and orthogonality of the language and the experience of the programmers.
- **Cost of Writing the Programs**. This is a function of the writability of the language, which depends on its closeness to the particular application.
- **Cost of Executing the Program.** It is of primary importance for large production programs that will be executed repeatedly. It is greatly influenced by language design. A language that requires many run lime type checks has slow execution speed.
- **Cost of Translation.** Programs are compiled many times while being debugged but arc executed only a few times, so it is necessary to have a fast and efficient compiler.
- **Cost of Implementation**. It is related to the language implementation system. A language whose implementation systems are either expensive or runs only on expensive hardware. will have a much smaller chance of ever becoming widely used.
- **Cost of Maintenance**. It includes both corrections and modifications to add new features. The cost of maintenance mainly depends on readability because maintenance is often done by individuals other than original author of the software. Poor readability can make the task extremely challenging.

SUBPROGRAMS

- In computer science, a subroutine or subprogram (also called procedure, method, function, or routine) is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code.
- As the name "subprogram" suggests, a subroutine behaves in much the same way as a computer program that is used as one step in a larger program or another subprogram.
- A subroutine is often coded so that it can be started ("called") several times and/or from several places during a single execution of the program, including from other subroutines, and then branch back (return) to the next instruction after the "call" once the subroutine's task is done.

There are two distinct categories of subprograms :

- Procedures
- Functions.

SYNTACTIC ELEMENT OF A LANGUAGE

The general syntactic style of language is set by the choice of the various syntactic elements. The different syntactic elements are:

Character Set

The choice of character set is one of the first to be made in designing language syntax. The character set is nothing but group of characters which are allowed in programming language

including special characters such as '\()\ '\n\ Each language may have its own character set or may language can share same set. There are many character sets used such as the ASCII set. each containing of different set of special characters in addition to the basic letters and digits. 'C character set is available on most input-output equipments. However, A PL character set cannot be used directly on most input-output devices.

Now a day the computer industry is getting more international. Different countries adds their own special characters in the set such as Spanish adds the tilde (~), French adds accets (') & other characters are present in some languages.

Identifiers

Identifier is a string of letters & digits beginning with a letter. These are names that are given to various program elements, such as variables, functions.

Operator Symbols

Most languages use the special characters + and - to represent the two basic arithmetic operations. Primitive operations may be represented entirely by special characters like APL. Alternatively identifiers may be used for all primitives like LISP e.g. PLUS. TIMES & so on. Some languages adopt some combination of special characters for some operators & identifiers. For others e.g. The FORTRAN .EQ & ** for equality and exponentiation respectively.

Keywords & Reserved Words

1) Keywords:

"A keyword is an identifier used as a fixed pan of the syntax of a statement."

Example:

'if' beginning a ‘C’ conditional statement,

'for' beginning a 'C’ iterative statement.

2) Reserved word:

"A keyword is a reserved word if it is not used as a programmer chosen identifier."

Most languages today use reserved words. Syntactic analysis during translation is made easier by using reserved words. For example. ‘C’ uses reserved words heavily while FORTRAN make syntactic analysis difficult. In this. Do & If statements are not actually reserved words used for iteration and condition checking. They can be used as a programmer-defined identifiers.

Noise Words

Noise words are optional words that arc inserted in statement to improve readability. For example, in COBOL, the 'GO TO’ statement transfer control within the program. It is written as,

GO TO label.

Where GO is required & TO is optional; it carries no information & is only used to improve readability.

Comments

Inclusion of comment in a program is an important part of its documentation.

A language may allow comments in several ways:

- Separate comments lines in the program, as in the “BASIC" - "REM" statement.
- Delimited by special markers, such as the ‘C’ ‘/*' & '*/'. The disadvantage is, a missing terminating delimiter on comments will turn the following statement (up to the end of next comments or end of the next comments or end of program) into comments & they will not be translated & executed.
- Beginning anywhere on a line but terminated by the end of the line, as the -- in Ada. // in 'C++’ ‘!’ T in FORTRAN90.

Blank Spaces

Rules on the use of blanks vary widely between languages. In ‘C’ blanks are used with characters & string data and separator between elements of statements. Other language use blanks as separators. In SNOBL04. the primitive operation of concatenation is represented by blanks, and the blanks are also used as a separator between elements of statements.

Delimiters & Brackets:

"Delimiters is a syntactic clement used to marks the beginning or end of some syntactic unit such as a statement or expression."

"Brackets are paired delimiters (example: parenthesis or begin— end pairs)."

Delimiters are used to enhance readability, to simplify syntactic analysis and remove ambiguities.

Free and Fixed-Field Formats:

Free Field Format:

"A language syntax is Free-Field if program statement may be written anywhere on an input line without regard for positioning on the line or for breaks between lines." Now a days, this syntax is used normally.

Fixed Field Format

"A Fixed-Field syntax uses the positioning on an input lines." Examples: Assembly language, COBOL.

Expressions

"Expressions are functions that access data objects in a program and return some value."

Expressions are the basic syntactic building block from which statements are built. In imperative languages like 'C' expressions form the basic operations that changes machine state during execution of each statement.

In applicative languages like ML or LISP, expressions form the basic sequence control that drives program execution.

Statement

Statements are the most prominent syntactic component in imperative languages. Their syntax has a critical effect on the overall regularity, readability & writability of the language. There are two main types of statements:

- Simple Statements: A simple statement contains no other embedded statement.
- Structured or Nested Statement: A Structured or Nested statement may contain embedded statement.

PROGRAMMING PARADIGM

Programming paradigm means style of programming. As programming matured, users and computer scientists found drawbacks and advantages in previous programming styles and evolved new programming styles.

We can classify programs into a combination of one or more programming paradigms:

- imperative programming,
- declarative programming,
- object-oriented programming,
- concurrent and distributed programming,
- visual programming,
- web-based programming,
- event-based programming,
- multimedia programming,
- agent-based programming, and
- synchronous programming.

Imperative Programming Paradigm

The basis of imperative programming is assertion or assignment statement that changes the state of the low-level von Neuman machine. A programmer manually translates the logic explicitly to tell the computer what to do. A variable in imperative programs is mapped onto a memory location in a computer, and the memory location can be modified repeatedly by using assignment statements. The effect of the assignment statement is that a new value is written into the memory location; the old value is lost. The advantage of assignment

statement is memory reuse, since the same memory location has been used for storing multiple value? multiple times.

Imperative programming paradigm was the first one to be developed in the late 1950s and 1960s in different versions of FORTRAN and in the early 1960s in the development block-structured languages such as ALGOL. Among imperative programming paradigms, four early languages were quite popular: FORTRAN, ALGOL and COBOL, and C, a descendant of Algol.

Early versions of FORTRAN were extensively used for scientific computing, made liberal use of jump statements, and did not support pointers. ALGOL developed a block-structured style of programming that supported control abstractions such as while-do loop and do-while loop that improved program comprehension. ALGOL also used pointers and "struct" that have been used to implement popular recursive data structures such as "linked-lists" and "trees". ALGOL is also known as the mother of modern-day imperative programming languages: most of the data abstractions and control abstractions described in ALGOL are still being used in modern-day languages.

C was originally a system programming language developed in Bell labs for writing the "UNIX" operating system. It was a subset of ALGOL 68 and gained popularity due to the popularity of Unix and its variations.

COBOL is a business programming language. It emphasizes on user friendliness, report writing, and handling financial data.

Many imperative programming languages such as ADA, FORTRAN, and COBOL kept on evolving by incorporating proven constructs. Newer versions of FORTRAN and COBOL have many features such as block-structured programming, recursive programming, stack-based implementation, string processing, pointers, structures, and object-oriented programming. With the passage of time, language designers identified the advantages and disadvantages of contemporary languages and incorporated the useful constructs. When a language evolves, care must be taken to keep it compatible with older versions, so that evolved programs can use older libraries, and programs written using older versions of languages can be compiled using compilers for newer versions.

Declarative Programming Paradigm

In declarative programming, control has been taken out of the program. There is an abstract machine that takes care of the control part implicitly. A declarative program consists of logic + abstraction at the programmer level. The notion of variable in declarative programs is quite different from the notion of variables in imperative languages. A variable in declarative program is a value holder; once a value has been assigned to a variable, it cannot be altered by the programmer. The advantage of assign-once property is (i) there is less possibility of side effects, as a called procedure (or function) cannot rewrite into memory space of the calling procedure (or function) and (2) the old values of variables can be retained and used if needed. There are also many disadvantages of the write-once property.

There are two major types of declarative programming languages: functional programming languages and logic programming languages. Functional programming languages arc based on the use of mathematical functions, and logic programming languages arc based on the use of predicate logic—Boolean logic combined with the notion of quantification, as explained in Chapter 2.

lisp, a popular language that combined functional programming wild sonic imperative constructs, was developed in the early 1960s for the implementation of artificial intelligence programming—a branch of computer science that tries to simulate human intelligence by computational means. *Prolog*, a popular logic programming language, was implemented in the 1970s for automated theorem proving and artificial intelligence programming.

Declarative programming languages have been traditionally used in artificial intelligence. The key features of declarative languages needed in artificial intelligence are as follows:

- The artificial intelligence systems need to dynamically develop and update knowledge. AI languages support abstractions for incorporating knowledge.
- Most declarative languages treat the program as a first class object, which means that programs can be built at run time, as data then transformed into a program that can be executed. Artificial intelligence needs this property to compile new knowledge.
- Declarative languages provide the capability of meta-programming—a program that reasons about another program in an abstract domain. This property of meta-programming is useful in developing reasoning and explanation capabilities in artificial intelligent systems.

Object-Oriented Programming Paradigm

As computers' memory size increased, executing large programs became possible. SIMULA was the first object-oriented language in the late 1960s. However, the notion of object-oriented programming caught on with the software development community in the early 1980s. As the program size grew, people started realizing the value of modularity and software reuse to develop even more complex software.

Many important interrelated concepts such as modularity, software reuse, off-the-shelf library, and information-hiding were conceived to provide ease of software development, maintenance, and evolution. Modularity means dividing large software programs into a set of interconnected modules, such that each one of them has clear functionality that does not overlap with the functionality of other modules, the major advantage of modular development is that the modification in one module does not adversely affect the functionality of other modules. Software reuse means that previously developed software can be reused by simply storing it in a tile and by importing the needed modules or subprograms from the archive when needed. Information hiding means to make the part of the program module implementation inaccessible to other modules if it is not needed for interaction with other modules.

In order to provide modularity and information hiding, the notion of objects and classes were discovered. An object has both data abstractions and methods—related functions or procedures needed to manipulate the data abstractions in the object. The information inside objects can be private, public, or protected. Public methods are available to the outside world; private methods are specific to the object providing information hiding; and protected methods arc visible to the objects in the subclasses of the current class. Information hiding is also related to the notion of modularity, as the hidden information cannot be used by other objects or classes. The notion of software reuse has been facilitated by the use of inheritance, where a subclass can inherit the methods from the parent class, and by the development of library of classes that can be developed and included in the software.

The first language to incorporate object-oriented programming was SIMULA. Smalltalk, developed by Xerox Pare and Eiffel, demonstrated the concept of objects. Since then many languages have been augmented with the object-oriented paradigm. For example, C++ is an integration of the language C, and the object-oriented paradigm, CLOS, is an integration of Lisp and the object-oriented programming paradigm. Many variations of Prolog have a library for object-oriented programming. Modern-day scripting languages, such as Python, Ruby, and PHP, have integrated object-oriented programming from their imperative ancestors. FORTRAN 2008 and COBOL 2002 are the latest evolution of FORTRAN and COBOL that have integrated the object-oriented programming paradigm with the earlier imperative versions. Java is a successor of C++ that integrates the Internet programming paradigm and the object-oriented programming paradigm. In recent years, X10 is a concurrent object-oriented programming language that has been developed by IBM researchers for high-level software development on massive parallel computers.

Concurrent Programming Paradigm

During the decade of 1980, as the software started getting bigger, the hardware technology was also changing very fast. Multiple fast processors were becoming available in a single computer, and the development in computer networking was allowing information exchange between multiple computers.

On the basis of this new advancement, independent subtasks could be mapped on separate processors to improve the execution efficiency of the programs. The development took two directions: (1) parallelizing compilers that could take a sequential program as input, and automatically transform it to a parallel version to execute concurrently and (2) incorporating high-level program constructs such as threads, forking, busy-wait-loop, and remote procedure calls in the existing languages. In all these constructs, a process—the active part of a program executing on a CPU—could start one or more concurrent subtasks to improve the execution efficiency.

The concurrent programming constructs are becoming common with the development of multicore-processor-based, modern-day personal computers. Modern-day programs and languages include many of these concurrency constructs to improve the execution efficiency of the programs. For example, the use of threads is quite common in modern-day languages,

either as an interface to existing thread libraries or as a built-in library. For example, an extensive library has been provided for C, C++, and Java.

Visual Programming Paradigm

The textual programming paradigm uses one dimension. However, we humans are very good in perceiving the notion of vicinity instead of just sequential one-dimensional programming. Most of the programming languages developed are textual and suffer from this limitation of sequentiality caused by the single dimension present in the textual representation of programs.

In the late 1980s, starting with earlier work in Smalltalk to provide user-friendly interfaces, the visual programming paradigm took multiple directions. Some suggestions were to free the programming languages using symbolic representation for different data abstractions and control abstractions at low-level programming. However, the effort to incorporate visual programming was limited to drag-and-drop programming to provide user-friendly interfaces and animations. More recently, visual programming has been used in languages such as C# for event-based programming: the symbols corresponding to events and object attributes are dragged and dropped to make a complex scenario of interacting objects. At the low level, these events and interacting objects are automatically translated to low-level textual version of the language. Similarly, visual educational languages such as Alice use visual programming to develop code for animation, and languages such as SMIL, VRML, and Java3D use visual programming for multimedia web-based on-demand presentation and animation.

The use of the visual programming paradigm for large-scale general purpose low-level programming has remained dormant due to the following: (1) the parsing difficulty in two-dimensional planes, (2) difficulty in human comprehension of large-scale programs in two-dimensional planes, and (3) difficulty of presenting large visual programs to human programmers. There is also no standardization of symbols used in visual programming.

Multimedia Programming Paradigm

The multimedia programming paradigm means integration of multiple modes of visualization: text, images, audio, video, and gestures. Humans interact with each other using all these cues. Without these cues, our communications with each others and our perception of objects and real-world phenomena would be incomplete. In the early 1990s, with the advancement of web-based programming and the development of comprehensive formats for audio, images, and video, it became possible to embed multimedia objects and video clips into programs for human visualization and perception. Video can be represented by a sequence of frames where each frame is a set of possibly interacting objects. There are languages, such as Alice and Virtual Reality Markup Language (VRML), that create 3D animated objects and motion to model virtual reality. In recent years, many such 3D modelling languages such as X3D and Java3D have been developing that integrate computation and 3D modelling for real-time animation over the Internet.

Web-Based Programming Paradigm

The advent of Internet in early 1990 has provided us with a tremendous capability for sharing data, images, audiovisuals, database, and mobile-code located remote web sites. It has also provided us with the capability of code and data mobility. If a remote resource does not want to share the code, it can compute the data at the source, and transmit the resulting data. On the other hand, if the server docs not want to get overloaded, it sends the code to the client to perform the computation at the client's end. Web-based programming has become a great engine for multimedia visualization and has made great impact on financial computing such as stock markets and banking. There are many Internet-based languages such as Java and SMIL, and web development languages such as PHP, JavaScript, and XML. XML has become a popular intermediate language for representing databases, computations, and animations over the Internet. Java has a popular intermediate-level abstract machine called JVM.

Java programs are interpreted using a JVM, which is similar to an assembly language for zero-address machines. A zero-address machine does not use a register or a memory address in the instruction. Rather it is a stack-based machine, where the operands arc placed on an evaluation stack, popped, evaluated, and pushed back on the stack. The reason for having a zero-address machine was that Java was designed to be implemented on any embedded computer in modern everyday devices such as microwaves, smart homes, and refrigerators. A zero-address machine is the most common abstract machine that could be run on all these embedded computers.

The problem with zero-address machines is that a high-level instruction is translated to many more instructions, wasting clock cycles in CPU. In order to speed up the execution, a new approach of just-in-time compilation has been developed, where known class libraries of the high-level language such as Java or C# arc compiled to the native binary code. When a high-level language program-fragment is translated to the low-level equivalent, then there are two possibilities: (1) translate to the compiled native binary code for faster execution or (2) use an abstract machine interpreter such as JVM or .NET (for a Microsoft Windows operating system). Justin-time compiled programs execute faster than interpreted codes on JVM.

Event-Based Programming Paradigm

Events are happenings that set up conditions that trigger some actions or computations. For example, clicking on a mouse is an event; moving the computer mouse over an image is an event; and an instrument reaching a threshold value is an event, the difference between traditional input-driven programming and event-based programming is that input-driven programming asks for some input, waits for the input, and then takes an action based upon the input value. In contrast, event-based programming never waits for any input from the external world. However, it responds to one or more events as soon as the event takes place. Event-based modelling can be used to model real-world phenomena, as events can become cause for further cascaded events resulting in additional actions. The concept of event-based programming has its roots in SIMULA developed during the early 1970s. However, it

became popular in recent years due to graphical and web-based interactions. More than one modern-day language such as C#, uses event-based programming to model animation and for graphical user interfaces.

Integration of Programming Paradigms

Modern-day languages do not subscribe to one programming paradigm; multiple programming paradigms are embedded in a language. For example, C++ supports the imperative programming paradigm and the object-oriented programming paradigm. Visual C++ also supports the visual programming paradigm. C# supports the imperative programming paradigm, the object-oriented programming paradigm, the event-based programming paradigm, and supports extensive visualization capabilities—a multimedia programming paradigm feature. Java has many characteristics of the imperative programming paradigm, the object-based programming paradigm, the web-based programming paradigm, the concurrent programming paradigm, and the event-based programming paradigm. Scala and Ruby integrate functional programming and object-oriented programming paradigms. The programming language X10 developed by IBM integrates the imperative programming paradigm, the object-oriented programming paradigm, and the concurrent programming paradigm. Modern scripting languages PHP and Python integrate imperative programming paradigms, shell-based programming, and object-oriented programming paradigms into one.

Each programming paradigm has both advantages and disadvantages. For example, the notion of objects in object-oriented programming, along with information hiding and the notion of modules, is suitable for large-scale software development and is being adopted by many older languages such as FORTRAN (in the latest version, Fortran 2008), that have evolved with time to incorporate many well-proven features such as stack-based implementation, recursion, and objects over a period of time. Similarly, COBOL (the latest version, COBOL 2002) has evolved to include objects. If a language stops evolving with time, programmers stop using the language for the lack of features, and old languages are replaced by new languages supporting the popular programming paradigms.

STRUCTURED PROGRAMMING

In structured programming design, programs are broken into functions (also known as modules, subprograms, subroutines, or procedures). Each function is designed to do a specific task with its own data and logic. Information can be passed from one function to another function through parameters. A function can have local data that cannot be accessed outside the function's scope. The results from different functions are synthesized in another function (main function). Many high level languages support structured programming.

Structured programming minimized the chances of one function affecting another. It supported to write clearer programs. It made global variables to disappear and to be replaced by local variables and parameters. Its organization helped to understand the programming logic easily. It also made debugging easier.

Functional abstraction was introduced with structured programming. Abstraction means the ability to look at something without bothering about its internal details. In a structured program, it is sufficient to know that a given function performs a specific task. How that task is performed is not important.

Drawbacks in Structured Programming

Though structured programming is a desirable feature, it also has certain drawbacks:

1. Data types are processed in many functions in a structured program. When changes occur in those data types, the corresponding changes must be made to every location that acts on those data types within the program. This is really a very time consuming task if the program is very large.
2. Let us consider the case of software development in which several programmers work as a team on an application. In a structured program, each programmer is assigned to build a specific set of functions and data types. Since different programmers handle separate functions that have mutually shared data types. Other programmers in the team must reflect the changes in the data types done by a programmer in the data types handled. Otherwise, it requires rewriting several functions.

BINDING AND BINDING TIME

Binding

"Binding is an association such as between an attribute and an entity or between an operation and a symbol."

Binding Time

'The lime al which binding lakes place is called Bin/ling Time." Binding can lake place at language design time, load time or run time.

Classes of Binding Times

Execution lime (run lime)

Many bindings are performed during program execution. This includes bindings of variables to their values, as well as the bindings of variables to particular storage locations. Two main categories are:

1. On entry to a subprogram or block:

In most languages, bindings are restricted to occur only at the time of entry to a subprogram or block during execution. For example: in C and C++, the binding of formal to actual parameters and the binding of formal to particular storage locations may occur only on entry to a subprogram.

2. At arbitrary points during execution:

Some bindings may occur at any point during execution of a program.

Example : The binding of variables to values through assignment, whereas some languages like LISP. Smalltalk and ML permit the bindings of names to storage locations to also occur at arbitrary points in the program.

Translation Time (Compile Time)

There are three different classes of translation time binding:

1. Binding chosen by the programmer:

While writing a program, programmer gives choice for variable names, types for variables, program statement structures and so on. That represents binding during translation.

2. Binding chosen by the translator:

Some bindings are chosen by translator. For example: the relative location of a data object in the storage allocated for a procedure, how arrays arc stored, how descriptors for the arrays, if any arc created all such decisions are made by the language translator.

3. Binding chosen by the loader:

A program usually consists of several subprograms that must be merged into a single executable program. The translator binds variables to addresses within the storage designated for each subprogram. However, this storage must be allocated actual addresses with the physical computer that will execute the program. This occurs during load time (also called link time).

Language Implementation Time

Some aspects of a language definition may vary between implementations. For example, the details associated with the representation of numbers and arithmetic operation may be determined by the underlying computer hardware. Hence, the program written in the language that uses a feature whose definition has been fixed at implementation time will not run on another implementation of the same language.

Language Definition Time

Most of the structure of programming language is fixed at language definition time. For example, different data types, data structure types, control elements, program structure and so on, are all fixed at language definition time.

Example

Illustrate variety of bindings and binding items for following statement:-

X = X + 10

Solution

1. Set of types for variable X :

   The set of allowable types for variable X is often fixed at language definition time.

2. Type of variable X :

The particular data type associated with variable X is often fixed at translation time through an explicit declaration in the program such as float X. In other languages, such as Smalltalk and Perl, the data type of X may be bound at execution time through assignment of a value of a particular type id X.

3. Set of possible values for variable X :

   The set of values of variable X is determined at language definition lime.

4. Value of variable X :

   At execution time, a particular value is hound to X. The assignment X = X + 10 changes the binding of X. replacing its old value by a new one i.e. ID plus old value.

5. Representation of constant 10:

   The choice of decimal representation in the program (i.e. using 10 for ten) is made at language definition lime, whereas the choice of the particular sequence of bits to represents 10 at execution time, usually made al language implementation time.

6. Properties of the operator +:

   The choice of symbol '+" to represent the addition operation is made at language definition time.

Example

```
int count ;
-----
count = count + 5;
```

- Set of possible types for count: - hound at language design lime.
- Type of count: - bound at compile lime.
- Set of possible values of count: - bound at language definition time.
- Value of count: - bound at execution time.
- Set of possible meanings for the operator symbol '+': - bound at language definition time.
- Meaning of the operator symbol +: - bound at compile lime.
- Internal representation of the literal 5." - bound at language implementation time.

ASSIGNMENT

Q.1. (AMIE W12, S15, 5 marks): List five high level programming languages. State one feature of each of the five high level languages.

Q.2. (AMIE S14, 7 marks): What is structured programming? Explain the relevant constructs using pseudo-code. Highlight the advantages and disadvantages of structured programming.

Q.3. (AMIE S11, 7 marks): What do you mean by orthogonality of a programming language? For "C" programming language, give examples of its orthogonality or otherwise.

Q.4. (AMIE S14, 7 marks): What are logical, syntactic and execution errors? Give one example of each. Which is most difficult to find and why?

Q.5. (AMIE S11, 7 marks): Explain the difference between static and dynamic binding by giving suitable examples.

Q.6. (AMIE S11, 6 marks): Explain pass by name parameter passing technique.

Q.7. (AMIE W11, 14, 5 marks): What is meant by binding time? Discuss various classes of binding time.

Q.8. (AMIE W13, 7 marks): Explain language defined binding, programmer defined binding and compiler defined binding for the following expression:

```
x = x + 5;
```

Q.9. (AMIE W11, 5 marks): Give a formula for determining the maximum number of bits required for storage of any value in the integer subrange M….N, where M and N are any two integers such that M < N.

Q.10. (AMIE W12, S15, 3 marks): Define and differentiate between a compiler and an interpreter.

Q.11. (AMIE W12, S15, 3 marks): Define programming paradigm. Explain main differences between two programming paradigms.

Q.12. (AMIE W14, 5 marks): What is meant by life time of a variable? Explain various categories of variables based their life time.