Unit 12 □ Programming Languages and Algorithm

Structure

12.0 Objectives

The objectives of the Unit are to :

- Explain the concept of programming language
- Examine the features of programming language
- Discuss classification of programming languages
- Present an overview of history of programming language
- Trace stages of programme translation
- Introduce the concept of Algorithm

12.1 Programming Languages

12.1.1 Introduction

Programming Language is an artifical language used to write a sequence of instructions that can be run by a computer. Similar to natural languages, such as Bengali, programming languages have a vocabulary, grammar, and syntax. However, natural languages are not suited for programming computers because their vocabulary and grammatical structure may be interpreted in multiple ways.The languages used to programme computers must have simple logical structures, and the rules for their grammar, spelling, and punctuation must be precise.

Programming languages allow people to communicate with computers. Once a job has been identified, the programmer must translate, or code it into a list of instructions that the computer will understand. A computer programme for a given task may be written in several different languages. Depending on the task, a programmer will generally pick the language that will involve the least complicated programme.

12.1.2 Definition

A programming language or computer language is a standardized communication technique for expressing instructions to a computer. It is a set of syntactic and semantic rules used to define computer programmes. A language enables a programmer to precisely specify what data a computer will act upon, how these data will be stored/ transmitted, and precisely what actions to take under various circumstances.

12.1.3 Features of programming language

Programming languages use specific types of statements, or instructions, to provide functional structure to the programme. A statement in a programme is a basic sentence that expresses a simple idea—its purpose is to give the computer a basic instruction. Statements define the types of data allowed, how data are to be manipulated, and the ways that procedures and functions work. Programmers use statements to manipulate common components of programming languages, such as variables and macros (mini-programmes within a programme).

Statements known as data declarations give names and properties to elements of a programme called variables. Variables can be assigned different values within the programme. The properties variables can have are called types, and they include such things as what possible values might be saved in the variables, how much numerical accuracy is to be used in the values, and how one variable may represent a collection of simpler values in an organized fashion, such as a table or array. In many programming languages, a key data type is a pointer. Variables that are pointers do not themselves have values; instead, they have information that the computer can use to locate some other variable—that is, they point to another variable.

Each programming language can be thought of as a set of formal specifications concerning syntax, vocabulary, and meaning. These specifications usually include :

- Data and Data Structures
- Instruction and Control Flow
- Reference Mechanisms and Re-use
- Design Philosophy

12.1.3.1 Data types

Internally, all data in modern digital computers are stored in binary format. The data typically represent information in the real world such as names, bank accounts and measurements and so the low-level binary data are organized by programming languages into these high-level concepts. The particular system by which data are organized in a programme is the *type system* of the programming language. Languages can be classified as *statically typed* systems, and *dynamically typed* languages.

With **statically typed languages,** there usually are pre-defined types for individual pieces of data (such as numbers within a certain range, strings of letters, etc.), and programmatically named values (variables) can have only one fixed type, and allow only certain operations : numbers cannot change into names and vice versa. Most mainstream statically typed languages, such as C, C++, C#, Java and Delphi, require all types to be specified explicitly; advocates argue that this makes the programme easier to understand.

Dynamically typed languages treat all data locations interchangeably, so inappropriate operations (like adding names, or sorting numbers alphabetically) will not cause errors until run-time-although some implementations provide some form of static checking for obvious errors. Examples of these languages are Objective-C, Lisp, Smalltalk, JavaScript, Tcl, Prolog, Python, and Ruby.

12.1.3.2 Data structures

Most languages also provide ways to assemble complex data structures from built-in types and to associate names with these new combined types (using arrays, lists, stacks, files).

Object oriented languages allow the programmer to define data-types called "Objects" which have their own intrinsic functions and variables (called methods and attributes respectively). A programme containing objects allows the objects to operate as independent but interacting sub-programmes : this interaction can be designed at coding time to model or simulate real-life interacting objects. This is a very useful, and intuitive, functionality. Languages such as Python and Ruby have developed as OO (Onject oriented) languages.

12.1.3.3 Instruction and control flow

Once data has been specified, the machine must be instructed how to perform operations on the data. Elementary statements may be specified using keywords or may be indicated using some well-defined grammatical structure.

Each language takes units of these well-behaved statements and combines them using some ordering system. Depending on the language, different methods of grouping these elementary statements exist. This allows one to write programmes that are able to cover a variety of input, instead of being limited to a small number of cases. Furthermore, beyond the data manipulation instructions, other typical instructions in a language are those used for control flow (branches, definitions by cases, loops, backtracking, functional composition).

12.1.3.4 Design philosophies

Since programming languages are artifical languages, they require a high degree of discipline to accurately specify which operations are desired. Programming languages are not error tolerant; however, the burden of recognizing and using the special vocabulary is reduced by help messages generated by the programming language implementation. There are a few languages which offer a high degree of freedom in allowing self-modification in which a programme re-writes parts of itself to handle new cases. Typically, only machine language, Prolog, PostScript, and the members of the Lisp family (Common Lisp, Scheme) provide this capability.

12.1.3.5 Compiled and interpreted languages

Computer programmes written in any language other than machine language must be either interpreted or compiled. An interpreter is software that examines a computer programme one instruction at a time and calls on code to execute the operations required by that instruction. This is a rather slow process. A compiler is software that translates a computer programme as a whole into machine code that is saved for subsequent execution whenever desired. Much work has been done on making both the compilation process and the compiled code as efficient as possible. When a new language is developed, it is usually at first interpreted. If the language becomes popular, it becomes important to write compilers for it, although this may be a task of considerable difficulty. There is an intermediate approach, which is to compile code not into machine language but into an intermediate language that is close enough to machine language that it is efficient to interpret-though not so close that it is tied to the machine language of a particular computer. It is use of this approach that provides the Java language with its computer-platform independence.

Programming languages generally fall into one of two categories : compiled or interpreted. With a compiled language, code is reduced to a set of machine-specific instructions before being saved as an executable file. With interpreted languages, the code is saved in the same format that you entered. Complied programmes generally run faster than interpreted ones because interpreted programmes must be reduced to

machine instructions at runtime. However, with an interpreted language you can do things that cannot be done in a compiled language. For example, interpreted programmes can modify themselves by adding or changing functions at runtime. It is also usually easier to develop applications in an interpreted environment because you don't have to recompile your application each time you want to test a small section.

12.1.4 History of programming languages

Charles Babbage is often credited with designing the first computer-like machines, which had several programmes written for them (in the equivalent of assembly language) by Ada Lovelace. However, in the 1940s the first recognizably modern, electrically powered computers were created.

Subsequent breakthroughs in electronic technology (transistors, integrated circuits, and chips) drove the development of increasingly reliable and more usable computers. The first widely used high level programming language was FORTRAN, developed during 1954-57 by an IBM team led by John W. Backus. It is still widely used for numercial work, with the latest international standard released in 2004.

In the late 1960s, the first object-oriented languages, such as SIMULA, emerged. Logic languages became well known in the mid 1970s with the introduction of PROLOG, a language used to programme artificial intelligence software. During the 1970s, procedural languages continued to develop with ALGOL, BASIC, PASCAL, C, and Ada. SMALLTALK was a highly influential object-oriented language that led to the merging of object-oriented and procedural languages in C++ and more recently in JAVA. Although pure logic languages have declined in popularity, variations have become vitally important in the form of relational languages for modern databases, such as SQL (Structured Query Language).

Dennis Ritchie and Brian Kernighan developed the C programming language, initially for DEC PDP-11 in 1970. Later with lead of Bjarne Stroustrup the programming language C++ appeared starting from 1985 as the Object oriented language vertically compatible with C. Sun Microsystems developed Java in 1995 which became very popular as the initial programming language taught at universities. Microsoft presented the C# programming language, bringing garbage collection, generic types, and introspection to a language that C++ programmers could learn easily, in 2001. There are other languages such as Python, Visual Basic, etc..

12.1.5 Classification

Programming languages can be classified as either low-level languages or highlevel languages. Low-level programming languages, or machine languages, are

the most basic type of programming languages and can be understood directly by a computer. Machine languages differ depending on the manufacturer and model of computer. High-level languages are programming languages that must first be translated into a machine language before they can be understood and processed by a computer. Examples of high-level languages are C, C++, PASCAL, and FORTRAN. Assembly languages are intermediate languages that are very close to machine language and do not have the level of linguistic sophistication exhibited by other high-level languages, but must still be translated into machine language.

12.1.5.1 Machnie Languages

In machine languages, instructions are written as sequences of 1s and 0s, called bits, that a computer can understand directly. An instruction in machine language generally tells the computer four things :

1. Where to find one or two numbers or simple pieces of data in the main computer memory (Random Access Memory, or RAM)
2. A simple operation to perform, such as adding the two numbers together,
3. Where in the main memory to put the result of this simple operation, and
4. Where to find the next instruction to perform.

While the computer in machine language eventually reads all executable programmes, they are not all programmed in machine language. It is extremely difficult to programme directly in machine language because the instructions are sequences of 1s and 0s. A typical instruction in a machine language might read 10010 1100 1011 and mean add the contents of storage register A to the contents of storage register B.

12.1.5.2 Assembly Language

Computer programmers use assembly languages to make machine-language programmes easier to write. In an assembly language, each statement corresponds roughly to one machine language instruction. An assembly language statement is composed with the aid of easy to remember commands. The command to add the contents of the storage register A to the contents of storage register B might be written ADD B, A in a typical assembly language statement. Assembly languages share certain features with machine languages. For instance, it is possible to manipulate specific bits in both assembly and machine languages. Programmers use assembly languages when it is important to minimize the time it takes to run a programme, because the translation from assembly language to machine language is relatively simple. Assembly languages are also used when some part of the computer has to be

controlled directly, such as individual dots on a monitor or the flow of individual characters to a printer.

12.1.5.3 High-Level Languages

High-level languages are relatively sophisticated sets of statements utilizing words and syntax from human language. They are more similar to normal human languages than assembly or machine languages and are therefore easier to use for writing complicated programmes. These programming languages allow larger and more complicated programmes to be developed faster. However, high-level languages must be translated into machine language by another programme called a compiler before a computer can understand them. For this reason, programmes written in a high-level language may take longer to execute and use up more memory than programmes written in an assembly language.

12.1.5.3.1 Classification of High Level Languages

High-level languages are commonly classified as procedure-oriented, functional, object-oriented, or logic languages. The most common high-level languages today are procedure-oriented languages. In these languages, one or more related blocks of statements that perform some complete function are grouped together into a programme module, or procedure, and given a name such as "procedure A." If the same sequence of operations is needed elsewhere in the programme, a simple statement can be used to refer back to the procedure. In essence, a procedure is just a mini-programme. A large programme can be constructed by grouping together procedures that perform different tasks. Procedural languages allow programmes to be shorter and easier for the computer to read, but they require the programmer to design each procedure to be general enough to be used in different situations.

Functional languages treat procedures like mathematical functions and allow them to be processsed like any other data in a programme. This allows a much higher and more rigorous level of programme construction. Functional languages also allow variables—symbols for data that can be specified and changed by the user as the programme is running—to be given values only once. This simplifies programming by reducing the need to be concerned with the exact order of statement execution, since a variable does not have to be redeclared, or restated, each time it is used in a programme statement. Many of the ideas from functional languages have become key parts of many modern procedural languages.

Object-oriented languages are outgrowth of functional languages. In object-oriented languages, the code used to write the programme and the data processed by

the programme are grouped together into units called objects. Objects are further grouped into classes, which define the attributes objects must have. A simple example of a class is the class Book. Objects within this class might be Novel and Short Story. Objects also have certain functions associated with them, called methods. The computer accesses an object through the use of one of the object's methods. The method performs some action to the data in the object and returns this value to the computer. Classes of objects can also be further grouped into hierarchies, in which objects of one class can inherit methods from another class. The structure provided in object-oriented languages makes them very useful for complicated programming tasks.

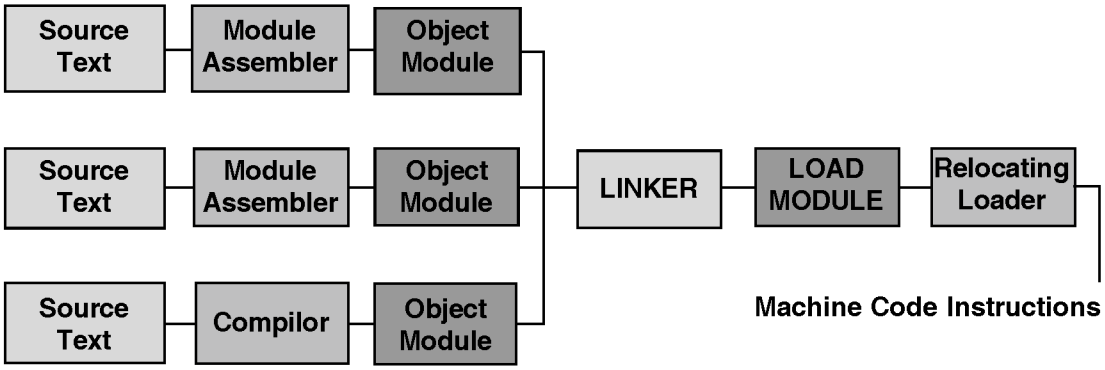
Logic languages use logic as their mathematical base. A logic programme consists of sets of facts and if-then rules, which specify how one set of facts may be deduced from others, for example :

If the statement X is true, then the statement Y is false. In the execution of such a programme, an input statement can be logically deduced from other statements in the programme. Many artificial intelligence programmes are written in such languages.

12.1.6 Programme Translation Sequence

In developing a software programme to accomplish a particular task, the programme developer chooses and appropriate language, develops the algorithm (a sequence of steps, which when carried out in the order prescribed, achieve the desired result), implements this algorithm in the chosen language (coding), then tests and debugs the final result. There is also a probable maintenance phase.

When you write a programme in a source language such as Pascal or C, the programme statements (in the source text file) needs to be converted into the binary bit-patterns which make sense to the target processor (the processor on which the software will be run). This process of conversion is called translation.



12.1.6.1 Assemblers

Assemblers are programmes that generate machine code instructions from a source code programme written in assembly language. The features provided by an assembler are :

- Allows the programmer to use mnemonics when writing source code programmes.
- Variables are represented by symbolic names, not as memory locations
- Symbolic code is easier to read and follow
- Error checking is provided
- Changes can be quickly and easily incorporated with a re-assembly
- Programming aids are included for relocation and expression evaluation.

A mnemonic is an abbreviation for a machine code statement. During the translation phase, each mnemonic is translated to an equivalent machine code instruction.

```
MOV AX, offh
is translated to the binary bit patterns
10111000 (this means MOV AX)
11111111 (this is ff hexadecimal)
00000000 (this is 0)
```

Assemblers also provide keywords called pseudo-ops. These keywords provide directions (hence they are also called assembler directives) to the assembler. Pseudo-ops do not generate machine instructions. The following pseudo-op

DB 'ab'

allocates and initializes a byte of storage for each character of the string, thus two bytes will be allocated, one initialized to the character 'a' whilst the other byte would be initialized to the character 'b'.

The assembler does not normally generate executable code. An assembler produces an object code file that must be further processed (linked) in order to generate a file that can be executed directly.

12.1.6.2 Interpreter

The source code programme is run through a programme called an interpreter. Each line of the programme is sent to the interpreter that converts it into equivalent machine code instructions. These machine code instructions are then executed. The next source line is then fetched from memory, converted and executed. This process is repeated till the entire programme has been executed.

Examples of interpreted languages are BASIC (Beginners All Purpose Symbolic Instruction Code) and Java.

12.1.6.3 Compiler

Compilers accept source programmes writtten in a high level language and produce object code programmes that are then linked with standard libraries to produce an executable file. Compilers generate code that is reasonably fast, but is target specific (it only runs on a particular computer system)

The source programme is
Written using an editor.
Most compiled
languages do not use
line numbers. The
example on the right is
a C programme.

```
# include <stdio. h>
main {
printf ("Hello world/n');
return 1;
}
```

Once the programme has been written using the appropriate source statements, it is then passed to a compiler that converts the entire programme into object code. The object code cannot be run on the computer system, so the object code file is then sent to a linker that combines it with libraries (other object code) to create an executable programme. Because the entire programme is converted to machine code, it runs very quickly.

12.1.6.4 Linker

The BASIC interpreter already has its own libraries for Input and Output (I/O), so BASIC programmes don't need linking. The source programme is converted directly into executable code.

Compiled languages (as well as assembled) need both linking and loading. The output of compilers and assemblers are stored in an intermediate format called object code. This is stored as a file on disk. The object code must be combined with other object code files or libraries (special object files) before execution.

The linker combines the programmes object code with the runtime object code files (for handling files, screen output, the keyboard etc) into an executable format. The types of files that exist at each phase of the programme translation sequence are,

myprog.c	source code programme
myprog.obj	object code produced by compiler
myprog.exe	executable file produced by linker

12.1.6.5 Loaders

It is normally the responsibility of the Operating System to load and execute files.

The part of the operating system that performs this function is called a loader. There are two types of loaders, relocating and absolute.

The absolute loader is the simplest and quickest of the two. The loader loads the file into memory at the location specified by the beginning portion (header) of the file, and and then passes control to the programme. If the memory space specified by the header is currently in use, execution cannot proceed, and the user must wait until the requested memory becomes free.

The relocating loader will load the programme anywhere in memory, altering the various addresses as required to ensure correct referencing. The decision as to where in memory the programme is placed is done by the Operating System, not the programmes header file. This is obviously more efficient, but introduces a slight overhead in terms of a small delay whilst all the relative offsets are calculated. The relocating loader can only relocate code that has been produced by a linker capable of producing relative code. A loader is unnecessary for interpreted languages, as the executable code is built up into the memory of the computer.

12.1.6.6 Locator

Programme locators convert the output of the linker (the executable file) into an absolute load format file. This type of file will eventually reside in specific memory locations, and is used to embed software into EPROM chips.

12.6.6.7 Cross Reference Utility (CRef)

These allow the programmer to generate a table that lists all symbols, labels, names, modules etc. Each ocurrence is listed, and generally the source programme is given line numbers to facilitate this process.

The Cref utility should detect data variables and assign symbols to them, presenting a variety of formats (by name, module etc). The Cref table is useful in debugging, as the programmer can ascertain in which modules a particular variable is referenced.

12.6.6.8 Disassembler

Disassemblers convert machine code instructions into mnemonic opcodes and operands, facilitating debugging at the machine code level. The more sophisticated disassemblers provide for

- Generation of symbols and labels
- Cross reference tools

- Disassembly of memory or disk files
- Output of disassembly to disk file
- Relocation information

12.6.6.9 Debuggers and Monitors

A monitor is a small programme that allows machine code access. A monitor provides, Debuggers provide much the same facilities as monitors, but generally provide a wider range of features,

- provision for HLL source debugging
- split screens, windowing
- reference by symbols, module names and labels
- radix changing
- Dynamic tracing of hardware interrupts
- Operating system calls and stack tracing

12.6.6.10 Cross Assembler

Cross assemblers allow a programmer to develop machine code programmes on one computer system for another system (target). In this way, a programmer can develop a machine code programme for a Macintosh computer system using an IBM-PC. The cross-assembler running on the PC generates the machine code instructions necessary for the Macintosh.

12.2 Algorithm

12.2.1 Introduction

The word algorithm comes from the name of the 9th century Persian mathematician Abu Abdullah Muhammad bin Musa al-Khwarizmi. The word algorism originally referred only to the rules of performing arthmetic using Arabic numerals but evolved into algorithm by the 18th century. The word has now evolved to include all definite procedures for solving problems or perfoming tasks.

The first case of an algorithm written for a computer was Ada Byron's notes on the analytical engine written in 1842, for which she is considered by many to be the world's first programmer. However, since Charles Babbage never completed his analytical engine the algorithm was never implemented on it.

Algorithms are essential to the way computers process information, because a computer programme is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an

algorithm can be considered to be any sequence of operations which can be performed by a computer system.

Typically, when an algorithm is associated with processing information, data is read from an input source or device, written to an output device, and/or stored for further use. Stored data is regarded as part of the internal state of the entity performing the algorithm.

For any such computational process, the algorithm must be rigorously defined, specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will almost always be cirtical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting 'from the top' and going 'down to the bottom', an idea that is described more formally by flow of control.

12.2.2 Definition

In computer science an algorithm is a finite set of well-defined instructions for accomplishing some task which, given an initial state, will terminate in a corresponding recognizable end-state. Algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison) until the task is completed. Correctly performing an algorithm will not solve a problem if the algorithm is flawed or not appropriate to the problem. Different algorithms may complete the same task with a different set of instructions in more or less time,space, or effort than others.

Nowadays, a formal criterion for an algorithm is that it is a procedure that can be implemented on a completely-specified Turing machine or one of the equivalent formalisms. Turing's initial interest was in the halting problem : deciding when an algorithm describes a terminating procedure. In practical terms computational complexity theory matters more : it includes the problems called NP-complete, which are generally presumed to take more than polynomial time for any (deterministic) algorithm. NP denotes the class of decision problems that can be solved by a non-deterministic Turing machine in polynomial time.

Some writers restrict the definition of algorithm to procedures that eventually finish. Others include procedures that could run forever without stopping, arguing that some entity may be required to carry out such permanent tasks. In the latter case, success can no longer be defined in terms of halting with a meaningful output. Instead,

terms of success that allow for unbounded output sequences must be defined. For example, an algorithm that verifies if there are more zeros than ones in an infinite random binary sequence must run forever to be effective. If it is implemented correctly, however, the algorithm's output will be useful : for as long as it examines the sequence, the algorithm will give a positive response while the number of examined zeros outnumber the ones, and a negative response otherwise. Success for this algorithm could then be defined as eventually outputting only positive responses if there are actually more zeros than ones in the sequence, and in any other case outputting any mixture of positive and negative responses.

Certain countries, such as the USA, allow some algorithms to be patented, provided a physical embodiment is possible.

12.2.3 Classes

There are many ways to classify algorithms. One way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include :

- Divide and conquer
- Dynamic programming
- The greedy method
- Linear programming
- Others

Another way to classify algorithms is by implementation. A recursive algorithm is one that invokes (makes reference to) itself repeatedly unitl a certain condition matches, which is a method comomon to functional programming. Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures where several processors can work on a problem at the same time. The various heuristic algorithms would probably also fall into this category, as their name (e.g. a genetic algorithm) describes its implementation.

References and Further Readings

1 2005 Algorithm (http://en.wikipedia.org/wiki/Algorithm). Visited last : 25/10/2005.

2 2005 Programming language (http://en.wikipedia.org/wiki/Programming language#Features_of_programming_language). Visited last : 21/10/ 2005.

3 Lamont (Michael). Algorithm and data structure (http://linux.wku.edu/~lamonml/ kb.html). Visited last : 25/10/2005.

12.3 Exercise

1. Discuss general features of computer programming languages.
2. Describe different generations of programming languages.
3. Discuss programme translation sequence.