**PRINCIPLES OF PROGRAMMING LANGUAGES**

## What is a Programming Language?

Programming languages are used to program digital computers. Prior to answering the above question, let's back up and address the question "what is a computer?"

*"A computer is a device that accepts information (in the form of digital data) and manipulates it for some result based on a program or sequence of instructions on how data is to be processed."*

Virtually all modern computers are digital, meaning that they operate on discrete (as opposed to continuous) values of numeric data, and the instructions are also discrete numbers. Analog computers also exist. They operate on signals that have a continuous (non-discrete) range of values. In this course, we'll only be concerned with digital computers.

Having limited our discussion to digital computers, the next issue to consider is the base of the number system used by the computer. Early digital computers, such as ENIAC, used a base-10 numbering system, that is, the numbering system used in everyday arithmetic, with an alphabet of digits ranging from 0 - 9. Hence, the term digital computer.

Building a computer with a base-10 numbering system is not easy. Electronic circuitry (e.g., memory cells) must be designed to deal with ten possible states, i.e., the digits 0 through 9. Computer designers soon realized that it is much easier to build computer circuitry that utilizes two states: "on" (or 1) and "off" (or 0), rather than ten states. Digital computers then became, strictly speaking, binary computers, since they use the
binary number system.

Binary numbers are almost universally used in modern electronic computers. With a binary computer, the "digital" data and the sequence of instructions referred to in the above definition consist of strings of 1s and 0s (corresponding to the "on" and "off" states in an electronic circuit).

Humans aren't used to dealing with a binary number system. Early programmers soon discovered that the process of programming is tedious and prone to error, partly due to the difficulty in representing complex computer instructions and data in the form of long strings of 1s and 0s. So, they began to look for a better way to program. Being programmers, they naturally looked to computers to solve their problem: they invented programming languages. Now, let's look at a definition of programming language:

*"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++, COBOL, FORTRAN, Ada, and Pascal. Each language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions."*

A high-level programming language like C++ allows a programmer to use the set of keywords, and a precise syntax, to write computer programs that are much closer to human language than strings of 1s and 0s. The computer is then used to translate the program into the machine language of 1s and 0s.

## Why Study Programming Languages?

The reasons for studying programming are

- ***Increased capacity to express ideas:*** Most programming languages have unique features and capabilities. The concepts of these features and capabilities often have wide application in other areas of information technology. As an example, many recent languages (e.g., C++, Java) are said to be "object-oriented." The object-oriented paradigm has been extended beyond the original domain of programming languages. As an example, object-oriented databases are now coming into vogue.

- ***As background for choosing an appropriate language:*** Software engineers are craft-persons, and view programming languages as tools. As in every craft, there is a "right" tool for every purpose. Whether an individual is a software engineer engaged in programming ("coding"), or an IT manager, it is crucial that he or she know and understand the available options and the strengths and weaknesses of the various programming languages.

- ***Improved ability to learn other languages:*** There are many similarities between programming languages. Once a software engineer has mastered a feature in one language (say, data structures), he or she is in a much better position to learn or appreciate similar features in another language.

- ***Better understanding of implementation issues:*** Knowing some details of how a programming language is implemented can make a software engineer or manager's decision on picking a programming language more informed. Realizing, for example, that the C language allows for very elegant and simple programs through the use of recursion is important, coupled with the understanding that recursion can result in slow and unreliable programs if not used carefully.

- ***Better ability to design new languages:*** Few programming languages have been designed and built from scratch. More typically, they are designed by experienced software engineers who are knowledgeable about the strengths and weaknesses of existing languages.

- ***Overall advancement in computing:*** Software engineering is an evolutionary process. Understanding what we have now will enable software engineers and IT managers to make incremental improvements to the field of computing.

## Influences on Language Design

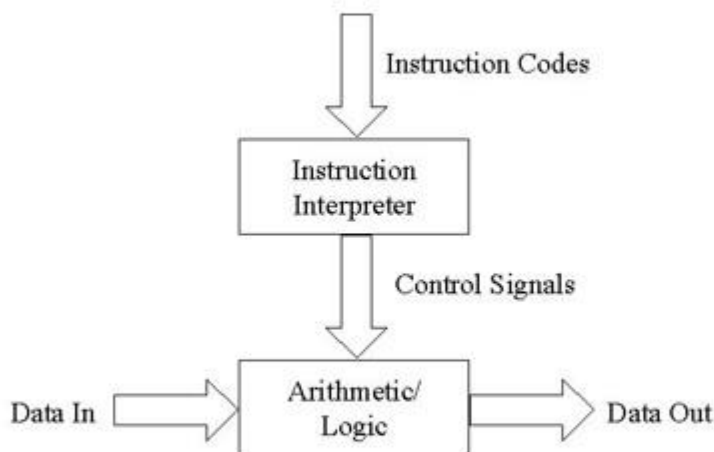
We started this session with a definition of "computer" and "programming language." Depending on your background, one's concept of what constitutes a computer or a programming language varies. A computer to one person might be a PC (based on the IBM/Intel architecture); to another person, it might be a SPARC 10000 or an IBM mainframe. Similarly, a programming language can range from primitive assembly language to Java. One might wonder it there's a "basic" computer and programming language.

It turns out that Alan Turing (1912-1954), a famous British mathematician, gave considerable thought to this question. He invented the concept of the Turing Machine -

*"A simple mechanical device consisting solely of a tape, a read/write head, and a finite-state machine. Turing was able to show that this machine is able to perform all the operations a person working with a logical system would be able to perform."*

A tape in the Turing Machine stores both the program instructions and data. The tape can move forward or backward, and can store results on the tape. Turing showed that this simple device can replicate any human computation. Later, he extended this concept to the "Universal" Turing Machine, i.e., that a Turing Machine can emulate any other computer. Hence the Turing Machine and its programming language can be considered "basic."

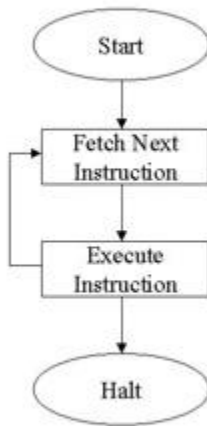
John von Neumann also had a profound effect on computing and programming languages. von Neumann is credited with developing the idea of controlling the operation of computer hardware through the manipulation of control signals (prior computers had to be physically rewired to change the computation being performed). von Neumann used the memory of the computer to store not only data, but also the sequence of control signals (i.e., the program) required to perform a calculation. In other words, he invented software programming.




Under the von Neumann architecture (Stallings fig. 3.1(b)) shown on the left, instructions are stored in a computer's memory. The instructions are sent to an ***instruction interpreter*** which translates them into control signals that determine how the ***arithmetic/logic unit*** (ALU) acts on the input data. The instructions in effect "morph" the ALU, making it appear to be a different piece of hardware every time a different instruction is received.

Instructions are sequentially "fetched" from memory, and then "executed," as shown below (Stallings book: ***Computer Organization and Architecture: Designing for Performance***

fig. 3.3). Early languages developed around the von Neumann architecture are known as Imperative Languages, and are characterized by variables (which model computer memory cells), assignment statements (which model the fetching operation), and iteration (the repetition of the process). The von Neumann architecture remains dominant today, and is often referred to as the von Neumann Bottleneck in that programs are executed in a serial (as opposed to parallel) manner, potentially limiting the speed of a computer.




In addition to the Turing and von Neumann influences on computing, the evolution of the computing industry has had profound effects on programming methodologies. As an example, the dramatically growing complexity of computer programs in the '60s necessitated a shift to so-called "structured" techniques. The lack of adequate control statements in early languages led to heavy use of "go to" statements, which then led to hard-to-maintain "spaghetti code." These and other shortcomings of Imperative languages led to the competing methodologies of Functional, Logical and Object-Oriented languages.

## Language Categories

Now that we've looked at the origin of Imperative languages, let's take a look at some of the important characteristics of the major language categories:

- ***Imperative Languages:*** Modeled after the von Neumann architecture, and characterized by variables (memory), assignment (movement of data), and iteration.
- ***Functional Languages:*** Based on the idea of a mathematical function, and characterized by no variables, no assignment, and no iteration. Functional languages depend heavily on recursion,
- ***Logical Languages:*** These are rule-based languages, characterized by a list of rules (in no particular order).
- ***Object-Oriented Languages:*** These languages emphasize data (rather than instructions), and are characterized by modules which integrate data structures that represent the elements of a problem with the operations needed to produce a solution.

## Programming Domains

Earlier, we stated that programming languages are tools, and that there is the right tool for every task. Let's attempt to categorize these tasks:

- ***Scientific Applications:*** The earliest computers (e.g., Eniac) were used for scientific applications, such as modeling and simulation, image processing, etc. These applications are characterized by extensive use of floating point and arrays, use of iteration ("loops"), and long computation times. Representative languages include FORTRAN ("FORmula TRANslation"), ALGOL ("ALGOrithimic Language").
- ***Business Applications:*** Business applications were first written in the 1950s, and include payroll, cost-accounting systems, and inventory control. These applications are

characterized by specialized data structures (e.g., monetary units), and the ability to generate complicated reports. COBOL is the premier example of a business-application language.

- ***Artificial Intelligence:*** AI applications were first written in the 1950s to address problems such as voice/image recognition, proving mathematical theorems, and providing the "brains" for "smart" weapons. AI languages are characterized by symbolic computation and list processing. Representative AI languages include Prolog and LISP.
- ***Systems Programming:*** Not all programs are user applications; programs are also needed to control the operation of the computing platform, e.g., job scheduling, input/output (I/O), security functions, etc. Such software constitutes a computer's operating system (OS). Also included in this category are the tools and utility programs used by system administrators to interact with the OS. System programs are characterized by their speed and efficiency, and by their ability to interact with low-level machine functions (e.g., I/O, interrupts, etc.). System programs are generally written in assembly language (which we'll describe later), C, PL/S (IBM machines), and BLISS (Digital machines).
- ***Scripting Languages:*** These are sets of commands, usually stored in a file execution. The autoexec.bat file on your PC is such a script that is run at boot time to initialize your machine. You can take a look at it by opening a DOS shell, and typing "type C:\autoexec.bat". The Common Gateway Interface (CGI) used for interactive Web pages is another example. Scripting languages are usually interpreted, not compiled (more about this later), and consequently run slowly. Unix scripts and PERL are representative scripting languages.
- ***Special-Purpose Languages:*** These languages are usually designed for a very specific purpose, such as statistical analysis or simulation. A few representative examples include SAS (Statistical Analysis System), GPSS (General Purpose Simulation System), NetRule (Analytical Network Simulation), Opnet (Discrete Event Network Simulation), Matlab (technical computing environment), MathCad (mathematical analysis), and Spice (electronic circuit simulation).

## Evaluating Programming Languages

Software engineers and IT managers need to know the strengths and weaknesses of programming languages. Many researchers have compiled lists of language evaluation criteria, e.g., writability, readability, portability, uniformity, extensibility, etc. Appleby and VandeKopple offer an extensive discussion of these "ilities." An early list generated by Barbara Liskov appears in Horowitz' book.

The somewhat abbreviated criteria presented by Sebesta (i.e., readability, writability, reliability, and cost are sufficient for our purposes:

- ***Readability:*** *"That quality of a programming language that enables a programmer to understand and comprehend the nature of a computation easily and accurately"* - Louden.

  Readability is strongly influenced by a number of factors, including simplicity (*"Everything should be as simple as possible, but no simpler."* - A.Einstein),

orthogonality, powerful control statements (eliminate the need for "GOTO"!), versatile data types and structures, and the syntactic design. Simplicity is dependent on the number of basic constructs available in a language. C, with four different ways to increment a simple integer variable, is not particularly simple to read:

```
count = count + 1;
count += 1;
count ++;
++count;
```

Conversely, overly simple languages (e.g., assembly language) are also difficult to read because far more instructions are needed; the Turing Machine language is the extreme case of simplicity.

The quality "orthogonality" means that language constructs can be combined in any meaningful way, and that the interaction of constructs, or context of use, should not cause unexpected restrictions on behavior (Louden book, p.52). As an example of a lack of orthogonality, the language C allows records to be returned from a function call, but not arrays. On the other hand, excessive orthogonality can lead to excessive complexity in a language, so there are trade-offs to consider.

Adequate control statements in a programming language also contribute to its readability. Control statements provide the capability of selecting among alternative control paths of statement execution. A primitive control capability, the "GO TO" statement, might be used like this:

```
IF (Index = 1) GO TO 10
IF (Index = 2) GO TO 20
IF (Index = 3) GO TO 10
IF (Index = 4) GO TO 20
```

Using the more sophisticated **case** statement results in easier-to-read code:

```
case Index of
    1,3:   begin
           (do something)
end
    2,4:   begin
           (do something else)
           end
else   error
end
```

A variety of data types and data structures also aids readability. As an example, C has five basic (primitive) data types: char, int, floating point, pointers, and void, but no Boolean type, so "true" is often represented as 1, and false as 0, e.g.,

```
sum_is_too_big = 0;
    if(sum > 100) sum_is_too_big = 1;
```

If a Boolean type were available in C, this could be written as:

```
sum_is_too_big = false;
 if(sum > 100) sum_is_too_big = true;
```

With this capability, later statements in the program could check whether sum_is_too_big is ***true*** or ***false***, rather than 0 or 1. This version using the Boolean type is clearly more readable

Finally, a language's syntax design has a major impact on readability. Sebesta identifies three things: identifier forms (i.e., variable names), the special words of a language, and the form and meaning of a language's statements as the factors affecting readability.

***Writability:*** *"Writability is a measure of how easily a language can be used to create programs for a chosen problem domain"* - Sebesta.
As with readability, simplicity and orthogonality affect writability. Complex languages are difficult to learn, and software engineers will end up using only a fraction of a complicated language's constructs. Conversely, overly simple languages (like the Turing Machine language) result in excessively long programs. Bottom line: an "optimum" language will balance simplicity and complexity, orthogonality and non-orthogonality.

A language that supports abstraction enhances writability. Using a simple operator, such as the plus (+) sign, to concatenate strings of variables is an example of abstraction, i.e.,

```
new_string = "Jomo" + " Kenyatta " + "University" + "Juja"
```

to concatenate four character strings into one.

Expressivity contributes to writability by making programs more compact.

***Reliability:*** Appleby and VandeKopple consider software to be reliable if it *"behaves as advertised and produces results the user expects."* Reliable software is easier to build if the language supports certain features such as type checking, exception handling, is readable and writable, and prohibits dangerous capabilities such as aliasing and pointers, which we'll discuss in later sessions.

Type checking is the process of evaluating expressions for type compatibility. A language with good type checking will not allow a program to attempt to add an integer to a character. Ada is a language with strong type checking; C is not.

Program exceptions are error conditions. Some languages have better provisions for exception handling than others. C++, Java, and Ada have exceptionally good exception handling; FORTRAN and C have little if any.

Some languages allow the same memory cell to have more that one variable name; this is called aliasing. FORTRAN allows a programmer to specify the same memory cell to be both an integer and a floating point variable. Usually, the motivation for aliasing is to overcome a language deficiency, but the consequence can be reduced reliability.

The impact of readability and writability on reliability is fairly obvious: writability directly improves reliability, and readability facilitates code reviews and walk-throughs which in turn enhances reliability.

## Implementing Programming Language

Earlier we made the observation that the purpose of programming language is to remove the programmer from the 1s and 0s of machine language to a higher level that is more understandable by humans. Hence, we need a mechanism to translate high-level language into machine language.

Generally, this translation is implemented in a series of steps using a layered approach. A layered approach has several advantages, e.g., it allows for error checking at each level, and it can be designed to allow many computer languages to run on the same hardware platform, as shown in Figure 1.2 of Sebesta.

Three implementation approaches are commonly used:

- ***Compilation:*** This is the process of translating a computer language (source code) into machine language (object code). Programs that are compiled usually run faster and more efficiently that those that use the other two approaches. The source code is written by a programmer using a combination of identifiers (variable names), operators (e.g., +, -, etc.), special words, punctuation, using a specific syntax (the form of the expressions, statements, program units, etc.). Compilation is accomplished by decomposing the source code through lexical and syntactical analysis to produce the machine code (strings of 1s and 0s). An important aspect of compilation is that it is completed prior to running (executing) the program. The overall process is illustrated in Sebesta Figure 1.3.
- ***Pure Interpretation:*** Program language interpreters also translate a high-level language into machine language, but they do it "on the fly," rather than completing it before program execution. The simplest interpreters interpret the source code line-by-line; after each line is interpreted, it is executed on the machine. Programs that are interpreted run much more slowly than compiled programs.
- ***Hybrid Implementations:*** Some programming languages, such as Java, use a hybrid approach. The high-level language is partially compiled into a simpler, intermediate language which is then interpreted. This approach gives a compromise between execution speed and portability (the ability to run on many different hardware platforms -- only the interpreter needs to be platform-specific.)

The above three approaches are ways to go from high-level language to machine language. Another important consideration is the software development environment, i.e., the collection of tools and utilities used to develop software.

A minimal software development environment consists of three components: a file system (to store high-level and machine-level code), an editor, and a compiler/linker. By way of example, VI (pronounced vee eye) is a bare-bones text editor available under the Unix environment, often used by C software developers.

More modern software development environments integrate a file system, an editor, a debugger, and a compiler/linker under a graphical user interface (GUI). Such systems greatly improve programmer productivity and SW quality, but may entail a steep learning curve. Microsoft's Visual C++ is an example of a sophisticated, GUI-based development environment.

## Ancient History

Augusta Ada Lovelace (1815-1852) probably has the honor of being the first programmer, working for Charles Babbage to program his mechanical "analytical engine." Babbage has the dubious distinction of being the first computer-system developer to incur a cost overrun.

Roughly a century later, during World War II, Konrad Zuse developed the programming language *Plankalkul* for use on an electromechanical computer. His language recognized several data types, including a single bit, integers, floats, arrays, and records.

Programming language evolved from primitive machine language (using binary numbers that represented instructions and data), to early attempts to automate/simplify the programmer's task, such as John Mauchly's short code for BINAC, and John Backus' Speedcode interpreter for the IBM 701. Short code, developed by John Mauchly in 1949, was implemented with a pure interpreter (we'll talk more about interpreters in a later session) simplified the programming process, but it was approximately 50 times slower than machine code

The work of Zuse, Mauchly, and Backus really laid the groundwork for modern programming language, starting with the invention of FORTRAN I in 1957. From that point in time, programming language rapidly evolved along several paths corresponding to the four language categories, as nicely shown in Figure 2.1 of Sebesta.

## Programming Language Category Examples

Let's now dive a bit deeper into the four categories of languages. For each category, we'll summarize the characteristic features, list representative languages, and give some specific language features/examples.

### Imperative Languages

***Characteristic Features:***

Imperative languages are characterized by their focus on *variables* (which relates to memory cells), by their use of *assignments* (which relates to moving data between the CPU and memory), and *iteration* (which relates to the fetch/execute von Neumann cycle that we discussed last session).

***Representative Languages:***

Representative imperative languages include FORTRAN ('**Formula Translating** System), ALGOL (**ALGO**rithmic **L**anguage), COBOL (**Co**mmon **B**usiness **O**riented **L**anguage), BASIC

(**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode), PL/1 (**P**rogramming **L**anguage 1), APL (**A** **P**rogramming **L**anguage), and SNOBOL (**S**tri**N**g **O**riented sym**BO**lic **L**anguage).

***Language Features/Examples:***

**FORTRAN:** John Backus started development of FORTRAN in 1954, with the goal of creating a high-level language that would be as efficient as hand-coded machine language, and that would eliminate coding errors; the first goal was largely achieved. FORTRAN I had no data type declarations; variables starting with the letter I, J, K, L, M, or N were implicitly integers, others were real. FORTRAN control constructs were very weak, e.g., branch instructions such as the "IF" statement. Later versions of FORTRAN improved many of its features, e.g., FORTRAN 77 introduced character-string handling and logical "IF" constructs, and FORTRAN 90 started to introduce object-oriented features. An important lesson from studying FORTRAN's evolution is how the best features of competing languages can be absorbed and improved on.

**ALGOL:** Algol development started in the late '50s. The goal of ALGOL was to provide a general, expressive language for describing algorithms for use in research and for practical applications. ALGOL introduced many new concepts, such as free-format, structured statements, begin-end blocks of code, type declarations of variables, recursion, and call-by-value (which we'll look at more closely in a later session). Being an imperative language, ALGOL introduced an explicit notation for variable assignment using colons, i.e., x := 3.0, meaning the value 3.0 is assigned to the variable x. ALGOL was also notable in that it was the first *portable* (able to run on multiple computer platforms) language, and it was the first language to be described by the *Backus Normal Form* notation, which we'll examine in the next session.

**BASIC:** Basic was designed at Dartmouth by John Kemeny and Thomas Kurtz in the early '60s, for use by liberal arts students. Basic was motivated by the (then) novel principle that user time is more valuable than computer time, leading to the concept of time-sharing. Basic's design drew heavily from FORTRAN. Descendants of Basic remain today (e.g., Visual Basic) as viable, useful programming language for some applications.

**C:** C was originally developed as a high performance language to be used for writing operating systems, e.g., UNIX. It was designed and implemented in the early '70s by Dennis Ritchie at Bell Labs. The classic book by Kernighan and Ritchie [1978] became the standard reference for C for many years. C is popular for its portability, its flexibility, and its power, but it's also criticized because its power and flexibility can lead to unreliability.

## Functional Languages

***Characteristic Features:*** With functional languages, computation is accomplished by applying functions to parameters. Functional languages lack the characteristics of imperative languages, i.e., variables, assignments, and iteration. However, since most machine architectures are von Neumann, functional languages are inefficient (i.e., run slowly).

***Representative Languages:*** Representative functional languages include LISP (**LIS**t **P**rocessor), Common LISP, Scheme, MetaLanguage (ML), Miranda, and Haskell.

***Language Features/Examples:***

**LISP:** LISP is a programming language usually associated with artificial intelligence (AI). LISP was developed by John McCarthy at MIT in the late '50s. Two main dialects of LISP have evolved: Common LISP (a combination of several dialects, it is now a large, complex language), and Scheme (which is characterized by its small size.

## Logic Languages

***Characteristic Features:*** Logic languages are characterized by being rule-based, with the rules specified in no particular order. Programs are written as a form of symbolic logic, and computations are accomplished through a process of logical inference. Like functional languages, logic languages tend to run inefficiently on machines built around the von Neumann architecture.

***Representative Language:*** Prolog (**Pro**gramming **Log**ic)

***Language Features/Examples:*** Programming in logical languages is 'declarative,' i.e., the user provides the computer with a description of objects, and the computer works out the details on how to solve the problem. Prolog was developed by Alain Colmerauer and Phillipe Rouseel of the University of Aix-Marseille, and Robert Kowalski of the University of Edinburgh in the early '70s. For a look how Prolog uses a process called *resolution*, jump ahead in Sebesta to page 629. The domain of problems for which Prolog is a good fit is clearly limited; scientific (numeric) calculations, for example, would be better accomplished with another language. Logic languages in general have proven to be inefficient, since they must spend time searching through many rules. Hence, logic languages are really only applicable to a few niches, such as AI and database engines.

## Object-Oriented Languages

***Characteristic Features:*** Object-oriented (OO) languages are characterized by their focus on abstract data types, their use of encapsulation and access control, and their use of inheritance.

The concept of encapsulation is in turn based on the concept of a class:

Class - *in object-oriented programming, a class is a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific instance of a class; it contains real values instead of variables.*

In object-oriented programming, the software engineer designs objects that contain data, and programs ("methods") for accessing and manipulating that data; hence, the data and methods are said to be "encapsulated." Good OO programming languages and practices preclude accessing the data in an object other than via the prescribed methods. This leads to potentially very important features of OO programs: reusability and maintainability.

***Representative Languages:*** Representative OO languages include Smalltalk, Ada, C++, and Java.

***Language Features/Examples:*** C++ is a good example of an OO programming language. C++ is currently in heavy use, due largely to its downward compatibility with C, and its high degree of portability. C++ implements most OO concepts, and is quite difficult to learn and to implement correctly. Since C++ is a superset of C, the reliability problems mentioned previously carry over.

## Wrap Up

We've covered a lot of ground this session while beginning to answer the question "What is a Programming Language." Along the way, we looked at the different program domains (scientific, business, etc.), categories (Imperative, Functional, Logic, and Object Oriented), some of the criteria used to evaluate languages.

Next session, we will move on to the *syntax* and *semantics* of programming language..

### Recommended Reading

***Programming Languages: Paradigm and Practice***
by Doris Appleby and Julius J. VandeKopple; 444 pages,
Published by McGraw-Hill Companies; 1997; ISBN 0-07-005315-4

***Computer Organization and Architecture: Designing for Performance***
by William Stallings; 682 pages
Published by Prentice Hall.; 1996; ISBN: 0-13-359985-X

***Fundamentals of Programming Languages, 2nd edition***
by Ellis Horowitz;
Computer Science Press (out of print), 1984

***Programming Languages : Principles and Practice***
by Kenneth C. Louden
Pws-Kent Series in Computer Science; 1993; ISBN: 0-53-493277-0

***The C Programming Language***
by Brian W. Kernighan, Dennis M. Ritchie; 274 pages
Published by Prentice Hall; 1988; ISBN: 0131103628