History and Use of Programming Languages
Andrés Gómez de Silva Garza

It's difficult to come to an agreement about the exact date in which a given programming language came into existence, because the development of any language is a lengthy process that does not always have a clear beginning or end. Even if these development dates were clear, when should one consider that the language has come into existence? When a development team formally decides, and announces to the world, that they want to develop a new language? When the first research paper describing the language appears in print? When the first version of the language is released for general use beyond the laboratory in which it was developed? When a "sufficiently large" (we would have to define what this means exactly) community of users has adopted the language for their programming needs?

So the dates mentioned in this document will necessarily be approximate (and it is important to note that not everyone will cite the same dates for the creation of any given language and therefore not everyone will be in agreement with the dates I have chosen). If we want to talk about "the first language," "the second language," etc., one also has to consider the fact that normally there are several languages under development at any given time and their development time-frames overlap, making it difficult to point out an exact order. One final point of contention when one talks about the history of programming languages is to decide exactly what a programming language is. Some people would consider that Ada Lovelace's algorithm for getting Charles Babbage's difference engine to make calculations is a program (expressed in a language involving the movements of mechanical gears and dials). Others might consider the Jacquard Loom (which could be programmed, using sequences of punched cards, to create textiles and tapestries) to be another example.

In this course we will focus exclusively on what are sometimes known as high-level programming languages, the types of programming languages that we are familiar with today. That is, programming languages that are meant to be used inside general-purpose electronic computers and in which the programming instructions are specified using a limited and highly structured pseudo-English (or similar).

If we want to go back to the history of programming languages of this kind, and if we also take into account the impact of a language (whether they are still in use or not, and/or whether their design influenced, or keeps influencing, the design of many languages that came later), we could list the following as "the first five important programming languages" (with approximate dates of creation, main developers, and place of creation listed):

1) FORTRAN (FORmula TRANslating system), 1957, John Backus @ IBM, San José, California. Purpose: mathematical, engineering, and scientific computation.

2) LISP (LISt Processing language), 1958, John McCarthy @ MIT. Purpose: interchangeability of code and data (important in Artificial Intelligence applications).

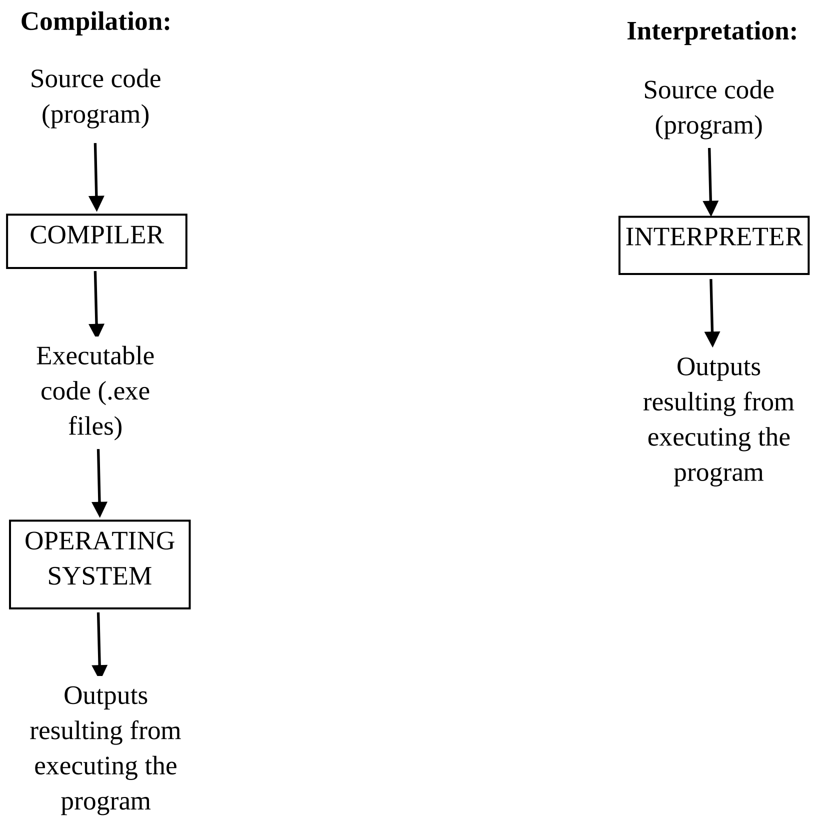
3) COBOL (COmmon Business-Oriented Language), 1959, Admiral Grace Hopper (and others) @ U.S. Department of Defense together with a committee of researchers from government, academia, and private industry. Purpose: business, finance, and administrative systems.

4) ALGOL (ALGOrithmic Language), 1960, Committee including Backus, McCarthy, Naur, Wirth, and others @ a meeting in Paris. Purpose: imperative, structured language.

5) BASIC (Beginner's All-purpose Symbolic Instruction Code), 1964, John Kemeny & Thomas Kurtz @ Dartmouth College, New Hampshire. Purpose: enabling students in non-technical fields to use (program) computers.

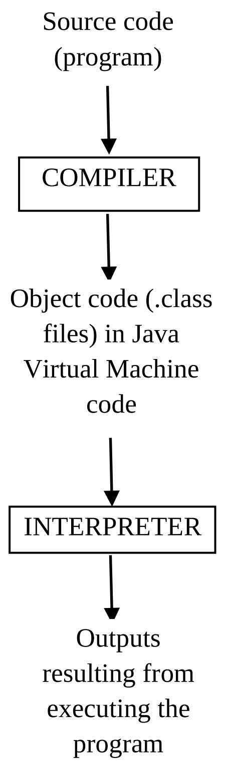
A chart showing the relationships between these and many other languages can be found in the file called ProgrammingLanguagesPoster.pdf (or HistoryOfProgrammingLanguages.pdf).

All of the languages that appear in the chart, and those that came afterwards, require translation. The programmer specifies instructions (source code) in some formal, systematic, limited, pseudo-English and these instructions have to be translated into machine code (consisting ultimately of 0's and 1's). Two mutually exclusive types of *translator* were used in the first few decades (and most languages in general provided one or the other method of translating, but not both): *compilers* and *interpreters*. A compiler, in its original conception, translates a program expressed in a high-level programming language to the machine language of a specific computer, perhaps passing through some intermediate versions of the program, such as an assembly language version of the program, before the user can execute the program. In Microsoft operating systems this ends up producing executable programs (.exe files), which can then be run/executed directly in the appropriate computer. An interpreter, in its original conception, translates a program (source code) expressed in a high-level programming language to the machine language of a specific computer, perhaps passing via an intermediate version of the program, as execution proceeds (i.e., interleaving translation with execution), without producing any executable (or other intermediate) file as a result of the process. *Assembly language* is a symbolic version of machine code, and is therefore easier to read for humans, and reflects the exact set of primitive instructions that are available to be executed by a particular processor. The following figure illustrates these two traditional forms of translation:


Compilation:
Source code (program)
COMPILER
Executable code (.exe files)
OPERATING SYSTEM
Outputs resulting from executing the program
Interpretation:
Source code (program)
INTERPRETER
Outputs resulting from executing the program

In a few modern languages (e.g., Java) the definitions of compiler and interpreter have changed. The Java compiler translates from the high-level language (i.e., from Java source code) to the machine language of a virtual machine (rather than a specific, existing, processor) before execution of the program is possible, producing object code (.class files). The Java interpreter translates a program expressed in the machine language of the Java virtual machine (.class files) into the machine language of a specific machine as execution proceeds, without producing any executable file as a result of the process. The Java interpreter does not take Java source code as its inputs. Both phases of translation are necessary in order to be able to run a Java program (although if the programmer is using an Integrated Development Environment like NetBeans, the fact that two phases are occurring might be transparent). If one wants to produce executable code (.exe files) corresponding to a Java program, (for instance to hide the source code from someone that we want to be able to use, but not to see, the instructions that form the program), one has to do even more subsequent translation (using tools such as JSmooth, Lanch4J, WinRun4J, or Nullsoft Scriptable Install System). The following figure illustrates this more modern approach to translating a program (and assumes that Java is the programming language):

Translation:


Source code (program)
COMPILER
Object code (.class files) in Java Virtual Machine code
INTERPRETER
Outputs resulting from executing the program

As an example of the different levels of languages we have been talking about, this is a fragment of a possible machine language program:

```
0000001010111100101 0
000000101111111001000
000000110011101010 00
```

The same program fragment expressed in assembly language is the following (note that we have the same number of instructions in the machine language and the assembly language versions of the program, but the machine language version was just 0's and 1's, whereas the assembly language version is symbolic):

```
LOAD I
ADD J
STORE K
```

The same program fragment in some high-level language could be expressed the following way (note that now the code is more compact, combines more things into one instruction, and is more legible to a human user, who does not need to know what basic instructions are available to the machine in which he/she intends to run the program):

```
k=i+j;
```

Programming Paradigms

Everyone agrees that there are different programming paradigms, and that most programming languages can be considered to be an example of one (or sometimes more than one) of them. The word *paradigm* in this context is used to refer to the type of approach that the programmer has to use to think about how to write the program that he/she needs to write. This refers to both the types of instructions that need to be used as well as the way in which those instructions need to be structured and expressed in the program. However, this definition is a little vague, so where there isn't 100% agreement is in deciding what paradigms exist. This leads often to situations in which one person considers that there are two programming paradigms but another person would consider them to be the same paradigm (or two sub-types of the same paradigm, or two historically separate but semantically similar/related examples of the same paradigm), and so on. For classification purposes, ideally all the paradigms would be mutually exclusive, so it would be easy to decide which paradigm a given language belongs to. In reality general-purpose languages have so many features (exactly because they are general-purpose) that the languages share characteristics belonging to multiple paradigms, though often (but not always) one main paradigm can be identified for a given language (usually based on what the language designer's explicit intentions were for the language, i.e., what the main focus of the research that produced the language was).

Here we briefly list and describe some approaches to programming that many people consider to be different programming paradigms (and mention some representative languages):

1) Functional programming: in a pure functional programming approach all computation is achieved through the definition and use of functions in the mathematical sense, and functions don't have any side-effects. Functions in the mathematical sense are transformations of sets of input values to output values. Programming in a functional language involves nesting function calls so that the input of one function can be the output of another one (executed previously). Note that this implies that programs are specified in a different way from the way in which they have to be specified in more traditional programming languages, where the order of execution (as well as the order in which instructions are specified) is sequential. The first functional programming language was LISP (1958).

2) Logic programming: in a pure logic programming approach all computation involves expressing knowledge using formal logic, and proving the validity of queries through the use of inference and logical operators. The first logic programming language was Prolog (1972).

3) Parallel/concurrent programming: a concurrent programming approach divides computation into modules that can be executed in parallel. Some of the best-known programming languages designed explicitly for concurrency are Modula-2 (1978) and Ada (1983), but Java (1995) also provides concurrency (via threads). While the approach to design/development that a programmer has to take is different if he/she has to think explicitly about concurrency, at the same time the concept of concurrency does not contradict the other paradigms we have mentioned, i.e., it is not mutually exclusive with the other approaches. An example is Concurrent Prolog (1987), which is a concurrent logic programming language. Because of this, some people might not consider parallel/concurrent programming to be a different programming paradigm.

4) Business-oriented programming: a business-oriented programming approach focuses on the type of data important for businesses, i.e., mainly simple numeric capabilities, highly-developed text processing and text formatting capabilities, etc. The first business-oriented programming language was COBOL (1959). Many people consider business-oriented programming to be just a special case of general-purpose programming, not really representing a different paradigm (approach to programming).

Some people would consider imperative programming (sometimes also known as, but sometimes considered to be different from, structured programming) to be a different programming paradigm. Imperative programming means that one programs by telling the computer what to do (the imperative tense in grammar is used when giving direct orders: "do this," "do that"), in the order in which one wants it to be done. This approach is not theoretically compatible with pure functional programming, where the programmer tells the computer the execution order indirectly (through nesting, rather than directly through sequences of instructions), or with pure logic programming, where the programmer only tells the computer what to do indirectly (by giving the computer knowledge which is used by the computer by applying general preprogrammed search procedures) rather than directly (by giving explicit instructions that tell the computer what to do with data step by step). Most programming languages nowadays support, and even promote, modularity, which is what structured programming implies: splitting programs into sub-parts that can be more easily designed, implemented, and reused/exchanged, than "spaghetti code." This is irrespective of whether the approach is functional, logical, imperative, or anything else. So in my opinion it is OK to consider imperative programming a separate programming paradigm, but it is not OK to consider structured programming a separate programming paradigm (or to equate it with imperative programming).

Most traditional programming languages are imperative, so imperative programming is the "normal," most common paradigm, and therefore of less interest intellectually. One long-lasting family of imperative programming languages, all of which also happen to be structured, is the branch of the programming languages family tree that includes ALGOL, Pascal, C, C++, and Java.

Some people would consider object-oriented programming to be a programming paradigm. A pure object-oriented programming approach focuses on defining and using data classes and objects, and structuring programs (again, through modularity) around these concepts. Some people consider this to just be a natural extension of imperative programming, but others consider it sufficiently different from non-object-oriented imperative programming to be a different paradigm. Most people consider the first object-oriented programming language to be Simula-67 (1967), though some people claim that Smalltalk (1972) is a more "pure" object-oriented language. In any case, object-oriented programming did not become popular or well-known until the late 1980's and early 1990's, when the need grew for teams of programmers to be able to easily write programs that manipulate highly complex data types, with non-trivial internal structure.