1. Basic Computing Concepts
(9) Computer Language Translation

9 Computer Language Translation

The term Source Code refers to program instructions in the way the programmer has written them e.g. in Java.

To execute the program, however, the programmer must translate it into machine language, the language that the computer understands. When the program is in machine code it is called Object Code.

Source code is the only format that is readable by humans. When you purchase programs, you usually receive them in their machine-language format. This means that you can execute them directly, but you cannot read or modify them.

Some software manufacturers provide source code, but this is useful only if you are an experienced programmer.

9.1 Generations of Programming Languages

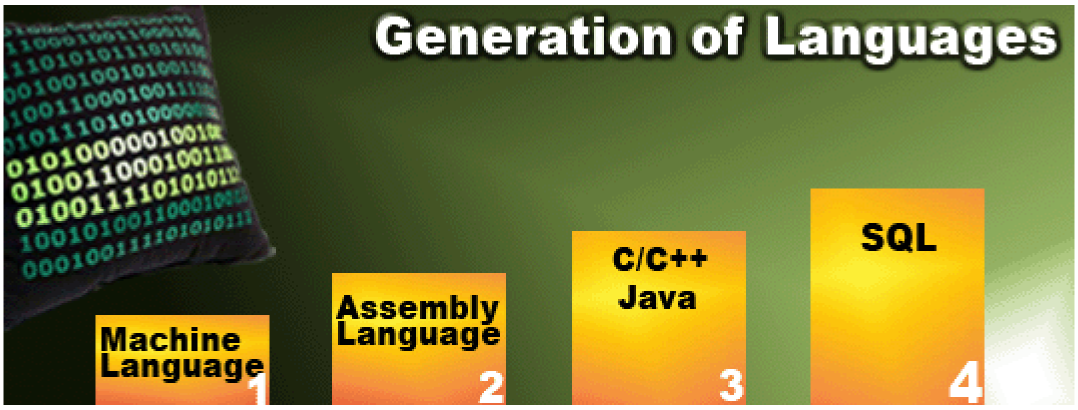


Diagram: Generation of Languages

9.1.1 The First Generation

The first generation programming languages is machine-level programming language. It consists of 1s and 0s.

Originally, no translator was used to compile or assemble the first-generation language. The first-generation programming instructions were entered through the front panel switches of the computer system.

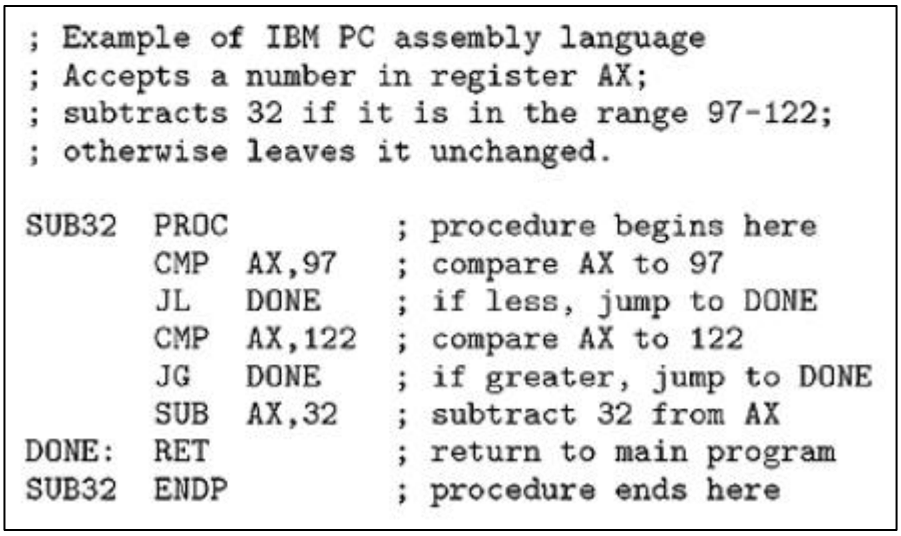
The main benefit of programming in a first-generation programming language is that code a user writes can run very fast and efficiently since it is directly executed by the CPU, but machine language is somewhat more difficult to learn than higher generational programming languages, and it is far more difficult to edit if errors occur. Furthermore portability is significantly reduced

- in order to transfer code to a different computer it needs to be completely rewritten since the machine language for one computer could be significantly different from another computer. Architectural considerations make portability difficult too. For example, the number of registers on one CPU architecture could differ from those of another.

9.1.2 The Second Generation

A second-generation programming language is a term usually used to refer to some form of assembly language. Unlike first-generation programming languages, the code can be read and written fairly easily by a human, but it must be converted into a machine readable form in order to run on a computer. The conversion process is simply a mapping of the assembly language code into binary machine code (the first-generation language). The language is specific to and dependent on a particular processor family and environment. Since it is the native language of a processor it has significant speed advantages, but it requires more programming effort and is difficult to use effectively for large applications.

Both first and second generation languages are called low-level languages because to use them one has to have knowledge of the architecture of the particular computer one is working on.



```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32   PROC            ; procedure begins here
        CMP   AX,97     ; compare AX to 97
        JL    DONE      ; if less, jump to DONE
        CMP   AX,122    ; compare AX to 122
        JG    DONE      ; if greater, jump to DONE
        SUB   AX,32     ; subtract 32 from AX
DONE:   RET             ; return to main program
SUB32   ENDP            ; procedure ends here
```

Diagram: A Program in Assembly Language

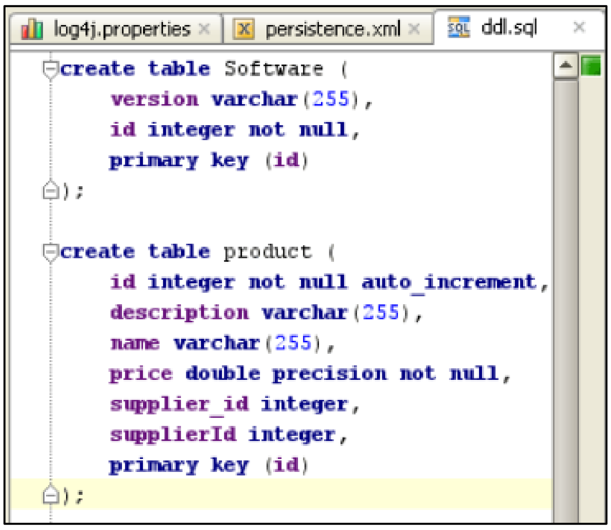
9.1.3 The Third Generation

A third generation language (3GL) is a programming language designed to be easier for a human to understand, including things like named variables. FORTRAN, ALGOL and COBOL are early examples of this sort of language. Most modern languages (BASIC, C, C++, Delphi, Java, and including COBOL, FORTRAN, ALGOL etc.) are third generation. Most 3GLs support structured programming.

From the third generation onwards, languages are considered to be high-level.

9.1.4 The Fourth Generation

A fourth-generation programming language (abbreviated 4GL) is a programming language designed with a specific purpose in mind, such as the development of commercial business software. Such languages arose after the introduction of modern, block-structured third-generation programming languages, which improved the process of software development. However, it was still considered by some to be frustrating, slow, and error prone to program computers. This led to the design of 4GLs. Many 4GLs are database languages.



```
create table Software (
     version varchar(255),
     id integer not null,
     primary key (id)
);

create table product (
     id integer not null auto_increment,
     description varchar(255),
     name varchar(255),
     price double precision not null,
     supplier_id integer,
     supplierId integer,
     primary key (id)
);
```

Diagram: A Program in SQL (a 4GL)

9.1.5 The Fifth Generation

A fifth-generation programming language (abbreviated 5GL) is a programming language based around solving problems using constraints given to the program, rather than using an algorithm written by a programmer.
While fourth-generation programming languages are designed to build specific programs, fifth-generation languages are designed to make the computer solve the problem for you. This way, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in artificial intelligence research. Prolog, OPS5, and Mercury are the best known fifth-generation languages.

In the 1990s, fifth-generation languages were considered to be the wave of the future, and some predicted that they would replace all other languages for system development, with the exception of low-level languages. Most notably, Japan put much research and money into their fifth-generation computer systems project, hoping to design a massive computer network of machines using these tools.

However, as larger programs were built, the flaws of the approach became more apparent. Today, fifth-generation languages have lost part of their initial appeal and are mostly used in academic circles.

9.2 Compilers and Interpreters

A compiler is a program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and translating it. Thus, a compiler differs from an interpreter, which analyses and executes each line of source code in succession, without looking at the entire program.

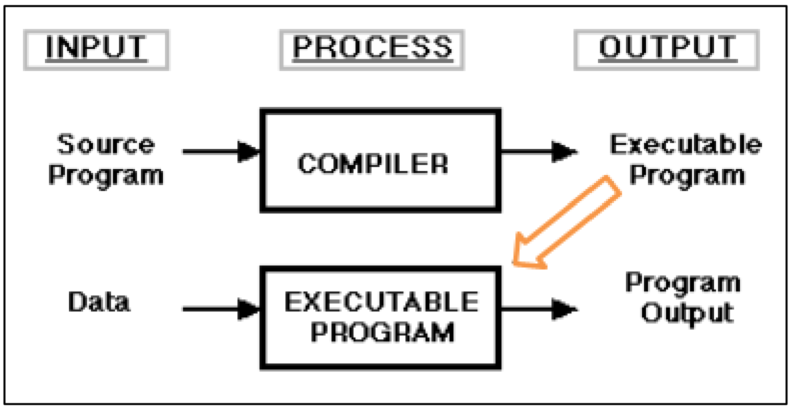


Diagram: A Compiler's Work

The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

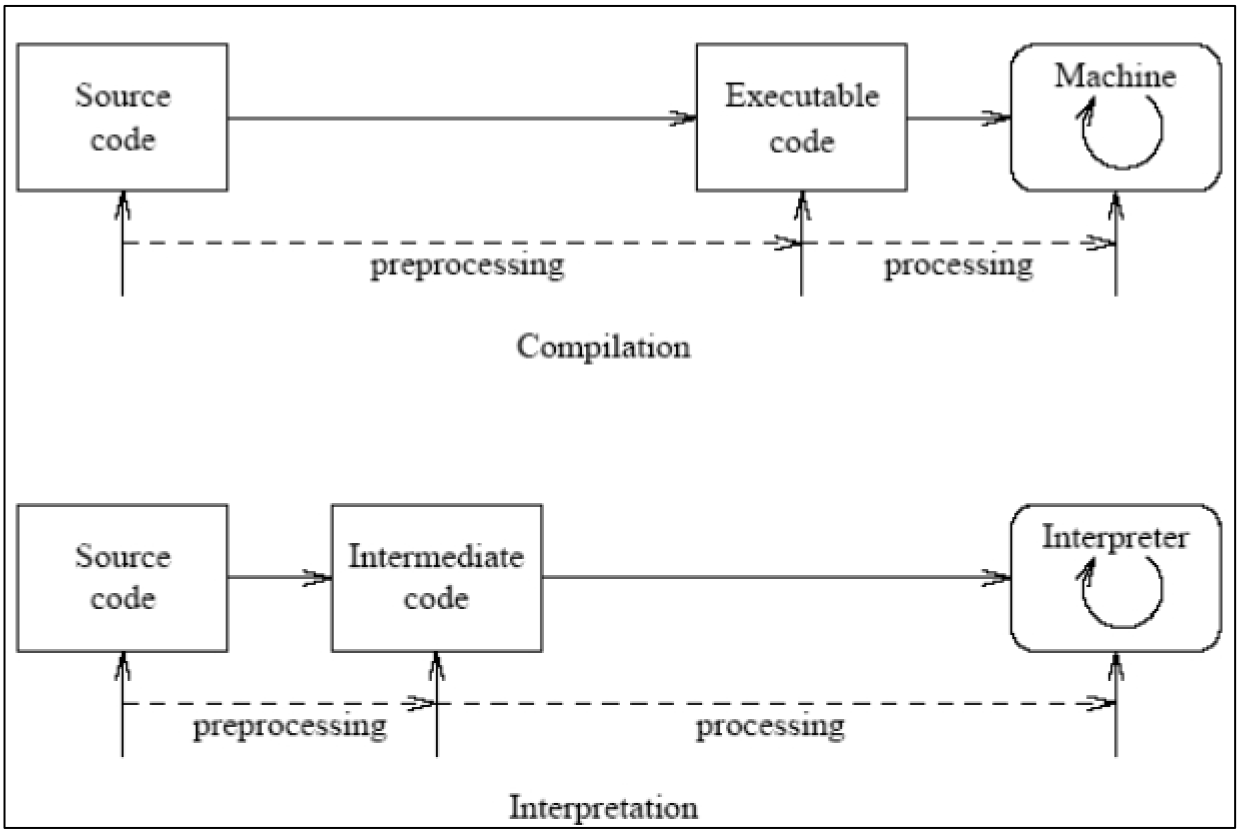


Diagram: Compiler and Interpreter

Because compilers translate source code into object code, which is unique for each type of computer, many compilers are available for the same language. For example, there is a FORTRAN compiler for PCs and another for Apple Macintosh computers. In addition, the compiler industry is quite competitive, so there are actually many compilers for each language on each type of computer. More than a dozen companies develop and sell C compilers for the PC.

9.2.1 Cross Compiler

A cross-compiler is a compiler that runs on one computer but produces object code for a different type of computer. Cross compilers are used to generate software that can run on computers with a new architecture or on special-purpose devices that cannot host their own compilers.

9.2.2 Just-in-time Compiler

In the Java programming language and environment, a just-in-time (JIT) compiler is a program that turns Java bytecode into instructions that can be sent directly to the processor. After you've written a Java program, the source language statements are compiled by the Java compiler into bytecode rather than into code that contains instructions that match a particular hardware platform's processor. The bytecode is platform-independent code that can be sent to any platform and run on that platform.

In the past, most programs written in any language have had to be recompiled, and sometimes, rewritten for each computer platform. One of the biggest advantages of Java is that you only have to write and compile a program once. The Java on any platform will interpret the compiled bytecode into instructions understandable by the particular processor. However, the virtual machine handles one bytecode instruction at a time. Using the Java just-in-time compiler (really a second compiler) at the particular system platform compiles the bytecode into the particular system code (as though the program had been compiled initially on that platform).

The just-in-time compiler comes with the virtual machine and is used optionally. It compiles the bytecode into platform-specific executable code that is immediately executed. Sun Microsystems suggests that it's usually faster to select the JIT compiler option, especially if the method executable is repeatedly reused.

9.2.3 Optimizing Compiler

This is a programming language compiler that enhances the performance and/or reduces the size of the resulting machine program. Optimizing compilers require multiple passes in order to analyse the entire program and maximize the reuse of code throughout.

9.3 Assemblers

An assembler is a program that takes basic computer instructions (in assembly language) and converts them into machine code.

In the earliest computers, programmers actually wrote programs in machine code, but assembler languages or instruction sets were soon developed to speed up programming. Today, assembler programming is used only where very efficient control over processor operations is needed. It requires knowledge of a particular computer's instruction set, however. Historically, most programs have been written in "higher-level" languages such as COBOL, FORTRAN, PL/I, and C. These languages are easier to learn and faster to write programs with than assembler language.

9.4 Java Virtual Machine

A newer idea in program preparation and portability is the concept of a virtual machine. For example, using the Java programming language, language statements are compiled into a generic form of machine language known as bytecode that can be run by a virtual machine, a kind of theoretical machine that approximates most computer operations. The bytecode can then be sent to any computer platform that has previously downloaded or built in the Java virtual machine. The virtual machine is aware of the specific instruction lengths and other particularities of the platform and ensures that the Java bytecode can run.

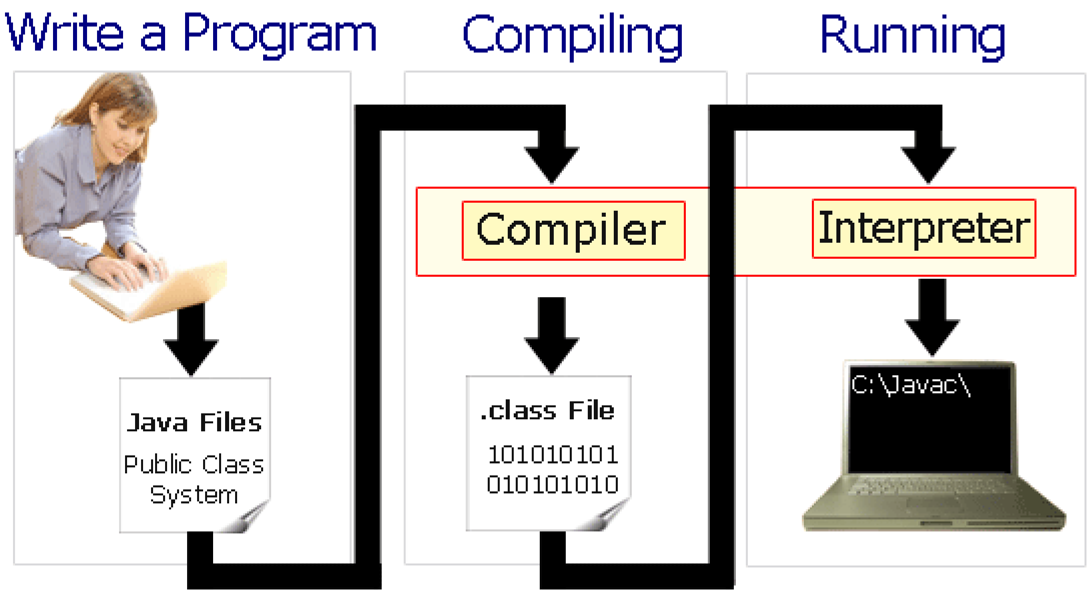


Diagram: Compiling and Executing a Program in Java

9.5 Linkers and Loaders

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.

In addition to combining modules, a linker also replaces symbolic addresses with real addresses. Therefore, you may need to link a program even if it contains only one module.

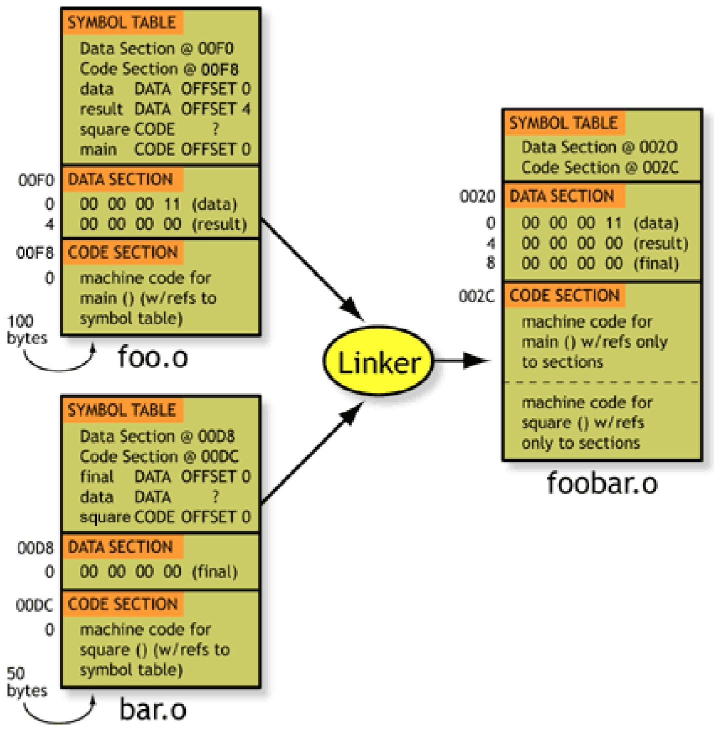


Diagram: A Linker

In an operating system, a loader is a component that locates a given program (which can be an application or, in some cases, part of the operating system itself) in offline storage (such as a hard disk), loads it into main storage, and gives that program control of the computer (allows it to execute its instructions).

A program that is loaded may itself contain components that are not initially loaded into main storage, but can be loaded if and when their logic is needed. In a multitasking operating system, a program that is sometimes called a dispatcher juggles the computer processor's time among different tasks and calls the loader when a program associated with a task is not already in main storage.

Diagram: A Loader