Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute.  
Assembly languages were soon developed that let the programmer specify instruction in a text format (e.g., ADD X, TOTAL), with abbreviations for each operation code and meaningful names for specifying addresses.  
The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference.  
For this purpose, algorithms are classified into orders using so-called Big O notation, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input.  
This can be a non-trivial task, for example as with parallel processes or some unusual software bugs.  
FORTRAN, the first widely used high-level language to have a functional implementation, came out in 1957, and many other languages were soon developed—in particular, COBOL aimed at commercial data processing, and Lisp for computer research.  
There are many approaches to the Software development process.  
Many applications use a mix of several languages in their construction and use.  
Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.  
For example, COBOL is still strong in corporate data centers often on large mainframe computers, Fortran in engineering applications, scripting languages in Web development, and C in embedded software.  
 After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug.  
Compiling takes the source code from a low-level programming language and converts it into machine code.  
To produce machine code, the source code must either be compiled or transpiled.  
A study found that a few simple readability transformations made code shorter and drastically reduced the time to understand it.  
 High-level languages made the process of developing a program simpler and more understandable, and less bound to the underlying hardware.