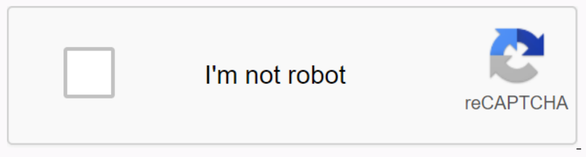

Continue

Assembly language and high level language pdf

Programming language with strong abstraction from hardware details Software language with strong abstraction from hardware details In computing is a high-level programming language with strong abstraction from computer details. Unlike a low level of programming languages, it can use natural language elements, is easier to use, or can automate (or even hide entirely) important areas of computer systems (e.g. memory management), making the process of developing a program simpler and comprehensible than using a lower-level language. The amount of predicted abstraction determines how high-level the programming language is. [1] In 60. [2] The case of autocoders is COBOL and Fortran. [3] The first high-level programming language dedicated to computers was Plankalkül, created by Konrad Zuse. [4] However, it was not implemented during his time, and his original contributions were largely isolated from other events due to The Second World War, in addition to the influence of the language on Heinz Rutishauser's Superplan language and to some degree algol. The first significantly expanded high-level language was Fortran, a machine-independent development of previous IBM Autocode systems. Algol, which was identified by committees of European and American computer scientists in 1958 and 1960, introduced rekuvri as well as nesting functions. It was also the first language with a clear distinction between value and name-parameters and their corresponding semantics. [5] Algol also presented several structured programming concepts, such as both meanwhile and if-then-other constructs, and its syntax was the first to be described in the formal notation – Backus-Naur form (BNF). During the approximately same period, Cobol introduced records (also called structural) and Lisp introduced lambda in the programming language for the first time. The High Level Language functions refer to a higher degree of abstraction from the machine language. Instead of dealing with registers, memory addresses and calling stacks, high-level languages address variables, matrixes, objects, complex arithmetic or boolean expressions, subprograms and functions, shuts, threads, locks and other abstract concepts of computing, with an emphasis on usability over optimal program efficiency. Unlike low-level languages, high-level languages have few language elements that translate directly into hardware codes. Other functions may also be present, such as string handling routines, object-oriented language functions, and file input/output. One thing to be abandoned about high-level programming languages is that these languages allow the programmer to be separated and separated from the machine. Unlike low-level languages such as assembly or machine language, high-level programming can and trigger a lot of data movements in the background without their knowledge. The responsibility and power of the executing instructions were surrendered to the machine by the programmer. Sentence abstraction High-level languages intend to provide functions that standardize common tasks, allow for rich debugging and perpetuate architectural agnosticism; while low-level languages often produce more efficient code by optimizing for a particular system architecture. Abstraction penalty is the cost that high software technicians pay for not being able to optimize capacity or use certain hardware because they do not exploit certain low-level architectural resources. High-level programs are features such as more generic data structures and operations, interpretation of startup time and intermediate code files; which often result in much more operations than necessary, more memory usage and a larger binary program size. [6] [7] [8] As a result, a code that must run particularly quickly and efficiently may require the use of a lower-level language, even if higher-level language facilitates coding. In many cases, critical parts of the programme can be manually encoded in a prefabricated language, leading to a much faster, more efficient or simply reliable operation of the optimized program. However, with the growing complexity of modern microprocessor architectures, well-designed complements for high-level languages often produce a code comparable to efficiency, with what most low-level programmers can produce by hand, and higher abstraction can provide more powerful techniques that deliver better overall results than their low-level counterparts in specific settings. [9] High-level languages are designed independently of a particular computer system architecture. This allows you to run a program written in that language on any computer with compatible Support for Interpreted or JIT. High-level languages can be improved as their designers develop improvements. In other cases, new high-level languages are developed from one or more others with the goal of combining the most popular constructs with new or improved features. An example of this is that it keeps compatibility back with Java, which means that programs and libraries written in Java will continue to be useful even if the software store switches to Scala; this facilitates the transition and life of such high coding indefinitely. By contrast, low-level programs rarely survive outside the system architecture for which they were written without a major audit. This is an engineering store for Abstraction Penalty. Relative importance This section does not list any sources. Help improve this section by adding citations to reliable sources. Material which is not pulled out may be challenged and disposed of. (October 2018) (Learn how and when to remove this template message) Examples Programming languages in active use today include Python, Visual Basic, Delphi, Perl, PHP, ECMAScript, Ruby, C#, Java and many others. Terms of high and low levels are inherently relative. A few decades ago, C language and similar languages were most often considered to be high level, supporting notions such as expression evaluation, parametric reursive functions and data types and structures, while the conical language was considered low level. Today, many C programmers can call it a low level because it does not have a large runtime system (without garbage collection, etc.), basically supports only scalar operations, and provides direct memory addressing. Therefore, it can be blended with the prefabricated language and the level of the CPU machine and micro-controls. Prefabricated language can be considered a higher level (but often still one-to-one if used without macros) to display hardware code because it supports concepts such as constants and (limited) terms, sometimes even variables, processes and data structures. The hardware code is inherently at a slightly higher level than the microcode or micro operations used internally in many processors. [10] Methods of execution This section does not list any sources. Help improve this section by adding citations to reliable sources. Material which is not pulled out may be challenged and disposed of. (October 2018) (Learn how and when to remove this template message) There are three general executing methods for modern high-level languages: Interpreted When an interpretation of a code written in a language, its syntax is read and then executed directly, without the degree of compilation. The program, called an interpreter, reads every program printout, follows the flow of the program, and then decides what to do and makes it. The hybrid interpreter and compiler will sit in the machine code and execute it; discard the hardware code if the line is re-executed. Interpreters are usually the simplest to perform language behaviour, compared to the other two variants listed here. When a code is written in a language, its syntax changes to the executing form before it is run. There are two types of compilation: Hardware Code Generation Some compilers assembled source code directly into the machine code. This is the original compilation mode, languages that are directly and completely transformed into machine-native code in this way can be called truly compound languages. See assembly language. Intermediate representations Where a code written in a language drawn up to an intermediate representation, that representation can be optimized or saved for subsequent executions without the need to re-read the source file. When an intermediate presentation is saved, it can be in a format such as bytecode. The intermediate representation must then be interpreted or further interpreted for enforcement. Virtual machines that direct or in the machine code, it blurred the once-clear distinction between intermediate displays and truly compound languages. Source-to-source code, translated or translated, written in a language, can be translated into a lower-level language for which source code compeltors are already common. JavaScript and C language are common goals for such translators. See CoffeeScript, Chicken Scheme, and Eiffel as an example. Specifically, the code C and C++ code created can be seen (as created from the Eiffel Language when using EiffelStudio IDE) in the EIFGENS directory of any assembled Eiffel project. In the Eiffel, the translated process is called transcompiling or transcompiled, and the Eiffel Translator as a compiler or compiler from source to source. Please note that languages are not strictly interpreted languages or compound languages. Instead, language behaviour uses interpretation or preparation. AlGOL 60 and Fortran, for example, were interpreted (although they were more typically composed). Similarly, Java points to problems using these tags in languages, not to run; Java is composed on bytecode, which is then executed either by interpreting (in Java virtual machine (JVM)) or compiling (usually with a flat-in-time compiler, such as HotSpot, again in JVM). Furthermore, the assembly, transcombing and interpretation is not strictly limited to the description of the composing artifact (binary executing or IL assembly). High-level computer architecture Another option is for your computer to run high-level language directly — the computer directly executes the HLL code. This is known as high-level computer architecture – computer architecture is designed to target a specific high-level language. For example, large Burroughs systems were target machines for ALGOL 60. [11] See also Computer Science Generational List of Programming Languages Low pressure programming languages High quality asembler Very high programming languages Categorical list of programming languages ^ References HThreads - RD Glossary ^ London, Keith (1968). 4, programming. Introduction to computers. 24 Russell Square London WC1: Faber and Faber Limited. P. 184. ISBN 0571085938. High-level programming languages are often called automatic codes and a processor program, kompiller. CS1 maint: location (link) ^ London, Keith (1968). 4, programming. Introduction to computers. 24 Russell Square London WC1: Faber and Faber Limited. P. 186. ISBN 0571085938. Two high-level programming languages that can be used here as examples to illustrate the structure and purpose of the auto-code are COBOL (Common Business Oriented Language) and FORTRAN (Translation forms). CS1 maint: location (link) ^ Giloi, Wolfgang, K. [de] (1997). Plankalkül Konrada Zusea: First High Non von Neumann Language. IEEE Annals of the History of Computing, vol. 19, No. 2, pp. 17-24, April-June, 1997. (abstract) ^ Although it did not have a conception of reference parameters, which could be a problem in some situations. Several successors, including AlgolW, Algol68, Simula, Pascal, Modula and Ada, therefore included reference parameters (the Related Language Family C allowed addresses as value parameters instead). ^ Surana P (2006). Meta-Compilation of Language Abstractions (PDF). Archived from the original (PDF) on 2015-02-17. Retrieved 2008-03-17. Cite journal requires |journal= (help) ^ Kuketayev. The Data Abstraction Penalty (DAP) Benchmark for Small Objects in Java. Archived from the original of 2009-01-11. Retrieved 2008-03-17. ^ Chatzigeorgiou; Stephanides (2002). Evaluation of the performance and power of object-oriented vs. procedural programming languages. In Blieberger; Strohmeier (eds.). Procedure - 7th International Conference on Reliable Software Technologies - Ada-Europe'2002. Springer. P. 367. ↑ Manuel Carro; José F. Morales; Henk L. Muller; Mr. Puebla; M. Hermenegildo (2006). High-level languages for small devices: case study (PDF). The process of the International Conference on Compilers, Architecture and Synthesis of Embedded Systems 2006. Acm. Hyde, Randall. (2010). There's no starch printing. ISBN 9781593273019. OCLC 635507601. ↑ Chu, Yaohan (1975), Concepts of High-Level Language Computer Architecture, High-Level Language Computer Architecture, Elsevier, pp. 1-14, doi:10.1016/b978-0-12-174150-1.50007-0, ISBN 9780121741501 External links - The WikiWikiWeb's article on Retrieved high-level programming languages programming from