Programming Languages

The theory of programming

Contents

Articles

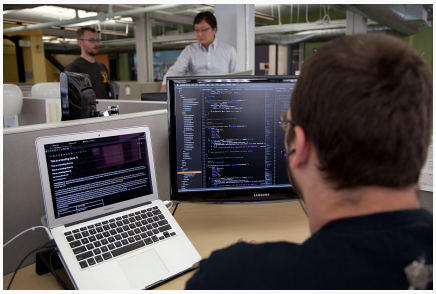
Theory

References

Article Licenses

Computer programming

Software development process



A software developer at work

Core activities

- Requirements
- Specification
- Architecture
- Construction
- Design
- Testing
- Debugging
- Deployment
- Maintenance

Methodologies

- Waterfall
- Prototype model
- Incremental
- Iterative
- V-Model
- Spiral
- Scrum
- Cleanroom
- RAD
- DSDM
- RUP
- XP
- Agile
- Lean
- Dual Vee Model
- TDD
- FDD
- DDD
- MDD

Supporting disciplines

• Configuration management • Documentation • Quality assurance (SQA) • Project management • User experience
Tools
• Compiler • Debugger • Profiler • GUI designer • Modeling • IDE • Build automation
• v • t • e [1]

Computer programming (often shortened to **programming**) is a process that leads from an original formulation of a computing problem to executable programs. It involves activities such as analysis, understanding, and generically solving such problems resulting in an algorithm, verification of requirements of the algorithm including its correctness and its resource consumption, implementation (commonly referred to as coding) of the algorithm in a target programming language, testing, debugging, and maintaining the source code, implementation of the build system and management of derived artefacts such as machine code of computer programs. The algorithm is often only represented in human-parsable form and reasoned about using logic. Source code is written in one or more programming languages (such as C, C++, C#, Java, Python, Smalltalk, JavaScript, etc.). The purpose of programming is to find a sequence of instructions that will automate performing a specific task or solve a given problem. The process of programming thus often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Overview

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process.

There is an on-going debate on the extent to which the writing of programs is an art form, a craft, or an engineering discipline. In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers." Because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming, and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance). However, representing oneself as a "professional software engineer" without a license from an accredited institution is illegal in many parts of the world.

Another on-going debate is the extent to which the programming language used in writing computer programs affects the form that the final program takes.Wikipedia:Citation needed This debate is analogous to that surrounding the Sapir–Whorf hypothesis[2] in linguistics and cognitive science, which postulates that a particular spoken language's nature influences the habitual thought of its speakers. Different language patterns yield different patterns

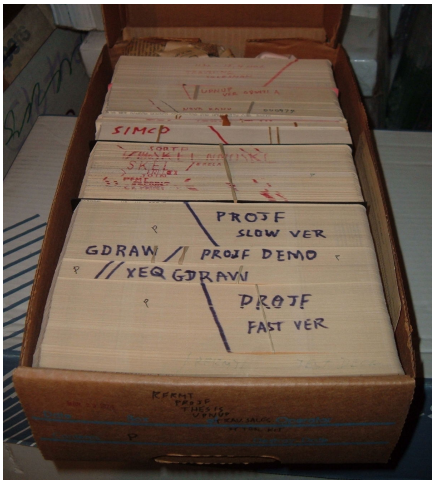
of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

History

See also: History of programming languages

Ada Lovelace created the first algorithm designed for processing by a computer and is usually recognized as history's first computer programmer.

Ancient cultures had no conception of computing beyond arithmetic, algebra, and geometry, occasionally aping elements of calculus (e.g. the method of exhaustion). The only mechanical device that existed for numerical computation at the beginning of human history was the abacus, invented in Sumeria circa 2500 BC. Later, the Antikythera mechanism, invented some time around 100 BC in ancient Greece, is the first known mechanical calculator utilizing gears of various sizes and configuration to perform calculations,[3] which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads. The Kurdish medieval scientist Al-Jazari built programmable automata in 1206 AD. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations, which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns. The Jacquard loom, which Joseph Marie Jacquard developed in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The first computer program was written for the Analytical Engine by mathematician Ada Lovelace to calculate a sequence of Bernoulli numbers. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through the Industrial Revolution.

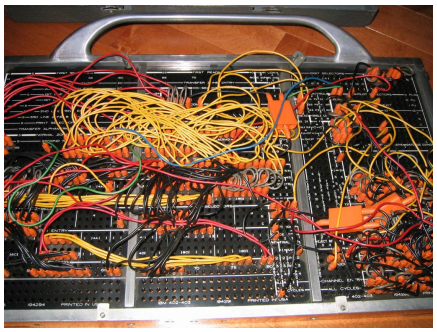


Data and instructions were once stored on external punched cards, which were kept in order and arranged in program decks.

In the 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for lists of instructions (not data) to drive programmed machines such as Jacquard looms and mechanized musical instruments. "After some initial trials with paper tape, he settled on punched cards..." To process these punched cards, first known as "Hollerith cards" he invented the keypunch, sorter, and tabulator unit record machines.[4] These inventions were the foundation of the data processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were several unit record calculators, such as the IBM 602 and IBM 604, whose control panels specified a sequence (list) of operations and thus were programmable machines.

The invention of the von Neumann architecture allowed computer programs to be stored in computer memory. Early programs had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a program in assembly language is usually more convenient, faster, and less prone to human error than using machine language, but because an assembly language is little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages.

Some of the earliest computer programmers were women. According to Dr. Sadie Plant, programming is essentially feminine—not simply because women, from Ada Lovelace to Grace Hopper, were the first programmers, but because of the historical and theoretical ties between programming and what Freud called the quintessentially feminine invention of weaving, between female sexuality as mimicry and the mimicry grounding Turing's vision of computers as universal machines. Women, Plant argues, have not merely had a minor part to play in the emergence of digital machines...Theirs is not a subsidiary role which needs to be rescued for posterity, a small supplement whose inclusion would set the existing records straight...Hardware, software, wetware-before their beginnings and beyond their ends, women have been the simulators, assemblers, and programmers of the digital machines.[5]



Wired control panel for an IBM 402 Accounting Machine

In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper. (A high-level language is, in very general terms, any programming language that allows the programmer to write programs in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The program text, or *source*, is converted into machine instructions using a special program called a compiler, which translates the FORTRAN program into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed,

including some for commercial programming, such as COBOL. Programs were mostly still entered using punched cards or paper tape. (See computer programming in the punch card era). By the late 1960s, data storage devices and computer terminals became inexpensive enough that programs could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought about newer programming languages that are more abstracted from the underlying hardware. Popular programming languages of the modern era include ActionScript, C, C++, C#, Haskell, PHP, Java, JavaScript, Objective-C, Perl, Python, Ruby, Smalltalk, SQL, Visual Basic, and dozens more. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code. However, high-level languages are still impractical for a few programs, such as those where low-level hardware control is necessary or where maximum processing speed is vital. Computer programming has become a popular career in the developed world, particularly in the United States, Europe, and Japan. Due to the high labor cost of programmers in these countries, some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities for programmers in less developed areas, particularly China and India.

Modern programming

Quality requirements

Whatever the approach to development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- Reliability: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- Robustness: how well a program anticipates problems due to errors (not bugs). This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, user error, and unexpected power outages.
- Usability: the ergonomics of a program: the ease with which a person can use the program for its intended purpose or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- Portability: the range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behavior of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.
- Maintainability: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.

- Efficiency/performance: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes careful management of resources, for example cleaning up temporary files and eliminating memory leaks.

Readability of source code

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability.

Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study[6] found that a few simple readability transformations made code shorter and drastically reduced the time to understand it.

Following a consistent programming style often helps readability. However, readability is more than just programming style. Many factors, having little or nothing to do with the ability of the computer to efficiently compile and execute the code, contribute to readability. Some of these factors include:

- Different indentation styles (whitespace)
- Comments
- Decomposition
- Naming conventions for objects (such as variables, classes, procedures, etc.)

Various visual programming languages have also been developed with the intent to resolve readability concerns by adopting non-traditional approaches to code structure and display.

Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. 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. 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.

Methodologies

The first step in most formal software development processes is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis. Many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process.

Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA.

A similar technique used for database design is Entity-Relationship Modeling (ER Modeling).

Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Measuring language usage

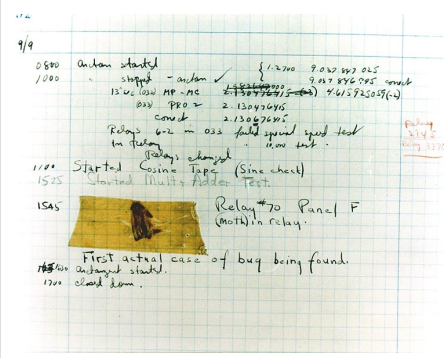
It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center,Wikipedia:Citation needed often on large mainframes, FORTRAN in engineering applications, scripting languages in Web development, and C in embedded applications), while some languages are regularly used to write many different kinds of applications. Also many applications use a mix of several languages in their construction and use. New languages are generally designed around the syntax of a previous language with new functionality added (for example C++ adds object-orientedness to C, and Java adds memory management and bytecode to C++, and as a consequence loses efficiency and the ability for low-level manipulation).

Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language,[7] the number of books sold and courses teaching the language (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

Debugging

Debugging is a very important task in the software development process since having defects in a program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static code analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, Code::Blocks, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.



The bug from 1947 which is at the origin of a popular (but incorrect) etymology for the common term for a software defect.

Programming languages

Main articles: Programming language and List of programming languages

Different programming languages support different styles of programming (called *programming paradigms*). 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. Ideally, the programming language best suited for the task at hand will be selected. 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. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones.

Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

> The details look different in different languages, but a few basic instructions appear in just about every language:

- Input: Gather data from the keyboard, a file, or some other device.
- Output: Display data on the screen or send data to a file or other device.
- Arithmetic: Perform basic arithmetical operations like addition and multiplication.

- Conditional Execution: Check for certain conditions and execute the appropriate sequence of statements.
- Repetition: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by shared libraries. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

Programmers

Main article: Programmer

See also: Software developer and Software engineer

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Debugging
- Documentation
- Integration
- Maintenance
- Requirements analysis
- Software architecture
- Software testing
- Specification

References

[1] http://en.wikipedia.org/w/index.php?title=Template:Software_development_process&action=edit

[2] Kenneth E. Iverson, the originator of the APL programming language, believed that the Sapir–Whorf hypothesis applied to computer languages (without actually mentioning the hypothesis by name). His Turing award lecture, "Notation as a tool of thought", was devoted to this theme, arguing that more powerful notations aided thinking about computer algorithms. Iverson K.E.," Notation as a tool of thought (http://elliscave.com/APL_J/tool.pdf)", *Communications of the ACM*, 23: 444-465 (August 1980).

[3] " Ancient Greek Computer's Inner Workings Deciphered (http://news.nationalgeographic.com/news/2006/11/061129-ancient-greece.html)". National Geographic News. November 29, 2006.

[4] U.S. Census Bureau: The Hollerith Machine (http://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html)

[5] Chun, Wendy. "On Software, or the Persistence of Visual Knowledge." Grey Room 18. Boston: 2004, pgs. 34-35

[6] James L. Elshoff, Michael Marcotty, Improving computer program readability to aid modification (http://doi.acm.org/10.1145/358589.358596), Communications of the ACM, v.25 n.8, p.512-521, Aug 1982.

[7] Survey of Job advertisements mentioning a given language (http://www.computerweekly.com/Articles/2007/09/11/226631/sslcomputer-weekly-it-salary-survey-finance-boom-drives-it-job.htm)

Further reading

- A.K. Hartmann, *Practical Guide to Computer Simulations* (http://www.worldscibooks.com/physics/6988.html), Singapore: World Scientific (2009)
- A. Hunt, D. Thomas, and W. Cunningham, *The Pragmatic Programmer. From Journeyman to Master*, Amsterdam: Addison-Wesley Longman (1999)
- Brian W. Kernighan, *The Practice of Programming*, Pearson (1999)
- Weinberg, Gerald M., *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold

External links

Library resources about **Computer programming**
• Online books (http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=wp&su=Computer+programming&library=OLBP) • Resources in your library (http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=wp&su=Computer+programming) • Resources in other libraries (http://tools.wmflabs.org/ftl/cgi-bin/ftl?st=wp&su=Computer+programming&library=0CHOOSE0)

- Software engineering (http://www.dmoz.org/Computers/Software/Software_Engineering/) at DMOZ
- Programming Wikia (http://code.wikia.com/wiki/Programmer's_Wiki)

History of programming languages

This article discusses the major developments in the history of programming languages [1] . For a detailed timeline of events, see: Timeline of programming languages.

Early history

The first programming languages predate the modern computer.

During a nine-month period in 1842-1843, Ada Lovelace translated the memoir of Italian mathematician Luigi Menabrea about Charles Babbage's newest proposed machine, the Analytical Engine. With the article she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the Analytical Engine, recognized by some historians as the world's first computer program.

Herman Hollerith realized that he could encode information on punch cards when he observed that train conductors encode the appearance of the ticket holders on the train tickets using the position of punched holes on the tickets. Hollerith then encoded the 1890 census data on punch cards.

The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, not only with words. For example, Alonzo Church was able to express the lambda calculus in a formulaic way. The Turing machine was an abstraction of the operation of a tape-marking machine, for example, in use at the telephone companies. Turing machines set the basis for storage of programs as data in the von Neumann architecture of computers by representing a machine through a finite number. However, unlike the lambda calculus, Turing's code does not serve well as a basis for higher-level languages—its principal use is in rigorous analyses of algorithmic complexity.

Like many "firsts" in history, the first modern programming language is hard to identify. From the start, the restrictions of the hardware defined the language. Punch cards allowed 80 columns, but some of the columns had to be used for a sorting number on each card. FORTRAN included some keywords which were the same as English words, such as "IF", "GOTO" (go to) and "CONTINUE". The use of a magnetic drum for memory meant that computer programs also had to be interleaved with the rotations of the drum. Thus the programs were more hardware-dependent.

To some people, what was the first modern programming language depends on how much power and human-readability is required before the status of "programming language" is granted. Jacquard looms and Charles Babbage's Difference Engine both had simple, extremely limited languages for describing the actions that these machines should perform. One can even regard the punch holes on a player piano scroll as a limited domain-specific language, albeit not designed for human consumption.

First programming languages

In the 1940s, the first recognizably modern electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned assembly language programs. It was eventually realized that programming in assembly language required a great deal of intellectual effort and was error-prone.

The first programming languages designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.[2]

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer.[3] Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. A programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

The second autocode was developed for the Mark 1 by R. A. Brooker in 1954 and was called the "Mark 1 Autocode". Brooker also developed an autocode for the Ferranti Mercury in the 1950s in conjunction with the University of Manchester. The version for the EDSAC 2 was devised by D. F. Hartley of University of Cambridge Mathematical Laboratory in 1961. Known as EDSAC 2 Autocode, it was a straight development from Mercury Autocode adapted for local circumstances, and was noted for its object code optimisation and source-language diagnostics which were advanced for the time. A contemporary but separate thread of development, Atlas Autocode was developed for the University of Manchester Atlas 1 machine.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype.[4] The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959.[5] Flow-Matic was a major influence in the design of COBOL, since only it and its direct descendent AIMACO were in actual use at the time.[6] The language Fortran was developed at IBM in the mid 1950s, and became the first widely used high-level general purpose programming language.

Other languages still in use today, include LISP (1958), invented by John McCarthy and COBOL (1959), created by the Short Range Committee, heavily influenced by Grace Hopper. Another milestone in the late 1950s was the publication, by a committee of American and European computer scientists, of "a new language for algorithms"; the *ALGOL 60 Report* (the "**ALGO**rithmic **L**anguage"). This report consolidated many ideas circulating at the time and featured three key language innovations:

- nested block structure: code sequences and associated declarations could be grouped into blocks without having to be turned into separate, explicitly named procedures;
- lexical scoping: a block could have its own private variables, procedures and functions, invisible to code outside that block, that is, information hiding.

Another innovation, related to this, was in how the language was described:

- a mathematically exact notation, Backus-Naur Form (BNF), was used to describe the language's syntax. Nearly all subsequent programming languages have used a variant of BNF to describe the context-free portion of their syntax.

Algol 60 was particularly influential in the design of later languages, some of which soon became more popular. The Burroughs large systems were designed to be programmed in an extended subset of Algol.

Algol's key ideas were continued, producing ALGOL 68:

- syntax and semantics became even more orthogonal, with anonymous routines, a recursive typing system with higher-order functions, etc.;
- not only the context-free part, but the full language syntax and semantics were defined formally, in terms of Van Wijngaarden grammar, a formalism designed specifically for this purpose.

Algol 68's many little-used language features (for example, concurrent and parallel blocks) and its complex system of syntactic shortcuts and automatic type coercions made it unpopular with implementers and gained it a reputation of being *difficult*. Niklaus Wirth actually walked out of the design committee to create the simpler Pascal language.

Some important languages that were developed in this period include:

- 1951 - Regional Assembly Language
- 1952 - Autocode
- 1954 - IPL (forerunner to LISP)
- 1955 - FLOW-MATIC (led to COBOL)
- 1957 - FORTRAN (First compiler)
- 1957 - COMTRAN (precursor to COBOL)
- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (forerunner to COBOL)
- 1959 - COBOL
- 1959 - RPG
- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (forerunner to C)
- 1964 - BASIC
- 1964 - PL/I
- 1966 - JOSS
- 1967 - BCPL (forerunner to C)

Establishing fundamental paradigms

The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period:

- **Simula**, invented in the late 1960s by Nygaard and Dahl as a superset of Algol 60, was the first language designed to support object-oriented programming.
- **C**, an early systems programming language, was developed by Dennis Ritchie and Ken Thompson at Bell Labs between 1969 and 1973.
- **Smalltalk** (mid-1970s) provided a complete ground-up design of an object-oriented language.
- **Prolog**, designed in 1972 by Colmerauer, Roussel, and Kowalski, was the first logic programming language.
- **ML** built a polymorphic type system (invented by Robin Milner in 1973) on top of Lisp, pioneering statically typed functional programming languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of "structured programming", which essentially meant programming without the use of Goto. This debate was closely related to language design: some languages did not include GOTO, which forced structured programming on the programmer. Although the debate raged hotly at the time, nearly all programmers now agree that, even in languages that provide GOTO, it is bad programming style to use it except in rare circumstances. As a result, later generations of language designers have found the structured programming debate tedious and even bewildering.

To provide even faster compile times, some languages were structured for "one-pass compilers" which expect subordinate routines to be defined first, as with Pascal, where the main routine, or driver function, is the final section of the program listing.

Some important languages that were developed in this period include:

- 1968 - Logo
- 1969 - B (forerunner to C)
- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1975 - Scheme
- 1978 - SQL (a query language, later extended)

1980s: consolidation, modules, performance

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Research in Miranda, a functional language with lazy evaluation, began to take hold in this decade.

One important new trend in language design was an increased focus on programming for large-scale systems through the use of *modules*, or large-scale organizational units of code. Modula, Ada, and ML all developed notable module systems in the 1980s. Module systems were often wedded to generic programming constructs---generics being, in essence, parametrized modules (see also polymorphism in object-oriented programming).

Although major new paradigms for imperative programming languages did not appear, many researchers expanded on the ideas of prior languages and adapted them to new contexts. For example, the languages of the Argus and Emerald systems adapted object-oriented programming to distributed systems.

The 1980s also brought advances in programming language implementation. The RISC movement in computer architecture postulated that hardware should be designed for compilers rather than for human assembly programmers. Aided by processor speed improvements that enabled increasingly aggressive compilation techniques, the RISC movement sparked greater interest in compilation technology for high-level languages.

Language technology continued along these lines well into the 1990s.

Some important languages that were developed in this period include:

- 1980 - C++ (as C with classes, renamed in 1983)
- 1983 - Ada
- 1984 - Common Lisp
- 1984 - MATLAB
- 1985 - Eiffel
- 1986 - Objective-C
- 1986 - Erlang
- 1987 - Perl
- 1988 - Tcl
- 1988 - Mathematica
- 1989 - FL (Backus)

1990s: the Internet age

The rapid growth of the Internet in the mid-1990s was the next major historic event in programming languages. By opening up a radically new platform for computer systems, the Internet created an opportunity for new languages to be adopted. In particular, the JavaScript programming language rose to popularity because of its early integration with the Netscape Navigator web browser. Various other scripting languages achieved widespread use in developing customized application for web servers such as PHP. The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many "rapid application development" (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention. More radical and innovative than the RAD languages were the new scripting languages. These did

not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain.Wikipedia:Citation needed Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web.

Some important languages that were developed in this period include:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1993 - Ruby
- 1993 - Lua
- 1994 - CLOS (part of ANSI Common Lisp)
- 1995 - Ada 95
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP
- 1996 - WebDNA
- 1997 - Rebol
- 1999 - D

Current trends

Programming language evolution continues, in both industry and research. Some of the current trends include:

- Increasing support for functional programming in mainstream languages used commercially, including pure functional programming for making code easier to reason about and easier to parallelise (at both micro- and macro- levels)
- Constructs to support concurrent and distributed programming.
- Mechanisms for adding security and reliability verification to the language: extended static checking, dependent typing, information flow control, static thread safety.
- Alternative mechanisms for modularity: mixins, delegates, aspects.
- Component-oriented software development.
- Metaprogramming, reflection or access to the abstract syntax tree
- Increased emphasis on distribution and mobility.
- Integration with databases, including XML and relational databases.
- Support for Unicode so that source code (program text) is not restricted to those characters contained in the ASCII character set; allowing, for example, use of non-Latin-based scripts or extended punctuation.
- XML for graphical interface (XUL, XAML).
- Open source as a developmental philosophy for languages, including the GNU compiler collection and recent languages such as Python, Ruby, and Squeak.
- AOP or Aspect Oriented Programming allowing developers to code by places in code extended behaviors.
- Massively parallel languages for coding 2000 processor GPU graphics processing units and supercomputer arrays including OpenCL

Some important languages developed during this period include:

- 2000 - ActionScript
- 2001 - C#
- 2001 - Visual Basic .NET
- 2002 - F#
- 2003 - Groovy
- 2003 - Scala
- 2007 - Clojure
- 2009 - Go
- 2011 - Dart
- 2012 - Rust

Prominent people

Some key people who helped develop programming languages (in alpha order):

- Joe Armstrong, creator of Erlang.
- John Backus, inventor of Fortran.
- Alan Cooper, developer of Visual Basic.
- Edsger W. Dijkstra, developed the framework for structured programming.
- Jean-Yves Girard, co-inventor of the polymorphic lambda calculus (System F).
- James Gosling, developer of Oak, the precursor of Java.
- Anders Hejlsberg, developer of Turbo Pascal, Delphi and C#.
- Rich Hickey, creator of Clojure.
- Grace Hopper, developer of Flow-Matic, influencing COBOL.
- Jean Ichbiah, chief designer of Ada, Ada 83
- Kenneth E. Iverson, developer of APL, and co-developer of J along with Roger Hui.
- Alan Kay, pioneering work on object-oriented programming, and originator of Smalltalk.
- Brian Kernighan, co-author of the first book on the C programming language with Dennis Ritchie, coauthor of the AWK and AMPL programming languages.
- Yukihiro Matsumoto, creator of Ruby.
- John McCarthy, inventor of LISP.
- Bertrand Meyer, inventor of Eiffel.
- Robin Milner, inventor of ML, and sharing credit for Hindley–Milner polymorphic type inference.
- John von Neumann, originator of the operating system concept.
- Martin Odersky, creator of Scala, and previously a contributor to the design of Java.
- John C. Reynolds, co-inventor of the polymorphic lambda calculus (System F).
- Dennis Ritchie, inventor of C. Unix Operating System, Plan 9 Operating System.
- Nathaniel Rochester, inventor of first assembler (IBM 701).
- Guido van Rossum, creator of Python.
- Bjarne Stroustrup, developer of C++.
- Ken Thompson, inventor of B, Go Programming Language, Inferno Programming Language, and Unix Operating System co-author.
- Larry Wall, creator of the Perl programming language (see Perl and Perl 6).
- Niklaus Wirth, inventor of Pascal, Modula and Oberon.
- Stephen Wolfram, creator of Mathematica.

References

[1] http://www.activatedesign.co.nz/history-of-computer-programming-languages

[2] Rojas, Raúl, et al. (2000). "Plankalkül: The First High-Level Programming Language and its Implementation". Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text) (http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm)

[3] Sebesta, W.S Concepts of Programming languages. 2006;M6 14:18 pp.44. ISBN 0-321-33025-0

[4] Hopper (1978) p. 16.

[5] Sammet (1969) p. 316

[6] Sammet (1978) p. 204.

Further reading

- Rosen, Saul, (editor), *Programming Systems and Languages*, McGraw-Hill, 1967
- Sammet, Jean E., *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969
- Sammet, Jean E., "Programming Languages: History and Future", *Communications of the ACM*, of Volume 15, Number 7, July 1972
- Richard L. Wexelblat (ed.): *History of Programming Languages*, Academic Press 1981.
- Thomas J. Bergin and Richard G. Gibson (eds.): *History of Programming Languages*, Addison Wesley, 1996.

External links

- History and evolution of programming languages (http://www.scriptol.com/programming/history.php).
- Graph of programming language history (http://www.levenez.com/lang/history.html)

Comparison of programming languages

Programming language comparisons

- General comparison
- Basic syntax
- Basic instructions
- Arrays
- Associative arrays
- String operations
- String functions
- List comprehension
- Object-oriented programming
- Object-oriented constructors
- Database access

- Evaluation strategy
- List of "Hello World" programs

- Web application frameworks
- Comparison of the Java and .NET platforms

- Comparison of individual programming languages
  - ALGOL 58's influence on ALGOL 60
  - ALGOL 60: Comparisons with other languages
  - Comparison of ALGOL 68 and C++
  - ALGOL 68: Comparisons with other languages
  - Compatibility of C and C++
  - Comparison of Pascal and Borland Delphi
  - Comparison of Object Pascal and C
  - Comparison of Pascal and C
  - Comparison of Java and C++
  - Comparison of C# and Java
  - Comparison of C# and Visual Basic .NET
  - Comparison of Visual Basic and Visual Basic .NET

- v
- t
- e [1]

Comparison of programming languages is a common topic of discussion among software engineers. Basic instructions of several programming languages are compared here.

Conventions of this article

The **bold** is the literal code. The non-bold is interpreted by the reader. Statements in guillemets (« … ») are optional. `Tab ␣` indicates a necessary indent (with whitespace).

Type identifiers

Integers

	8 bit (byte)		16 bit (short integer)		32 bit		64 bit (long integer)		Word size		Arbitrarily precise (bignum)
	Signed	Unsigned	Signed	Unsigned	Signed	Unsigned	Signed	Unsigned	Signed	Unsigned	
Ada[2]	`range -2**7 .. 2**7 - 1`[j]	`range 0 .. 2**8 - 1`[j] or `mod 2**8`[k]	`range -2**15 .. 2**15 - 1`[j]	`range 0 .. 2**16 - 1`[j] or `mod 2**16`[k]	`range -2**31 .. 2**31 - 1`[j]	`range 0 .. 2**32 - 1`[j] or `mod 2**32`[k]	`range -2**63 .. 2**63 - 1`[j]	`mod 2**64`[k]	`Integer`[j]	`range 0 .. 2**Integer'<wbr/>Size - 1`[j] or `mod Integer'<wbr/>Size`[k]	N/A
ALGOL 68 (variable-width)	`short short int`[c]	N/A	`short int`[c]	N/A	`int`[c]	N/A	`long int`[c]	N/A	`int`[c]	N/A	`long long int` [a][g]
									`bytes` & `bits`		
C (C99 fixed-width)	`int8_t`	`uint8_t`	`int16_t`	`uint16_t`	`int32_t`	`uint32_t`	`int64_t`	`uint64_t`	`int`	`unsigned int`	N/A
C++ (C++11 fixed-width)											
C (C99 variable-width)	`signed char`	`unsigned char`	`short`[c]	`unsigned short`[c]	`long`[c]	`unsigned long`[c]	`long long`[c]	`unsigned long long`[c]			
C++ (C++11 variable-width)											
Objective-C	`signed char`	`unsigned char`	`short`[c]	`unsigned short`[c]	`long`[c]	`unsigned long`[c]	`long long`[c]	`unsigned long long`[c]	`int` **or** `NSInteger`	`unsigned int` **or** `NSUInteger`	
C#	`sbyte`	`byte`	`short`	`ushort`	`int`	`uint`	`long`	`ulong`	`IntPtr`	`UIntPtr`	`System.Numerics<wbr/>.BigInteger` (.NET 4.0)
Java	`byte`	N/A		`char`[b]		N/A		N/A	N/A	N/A	`java.math<wbr/>.BigInteger`
Go	`int8`	`uint8` or `byte`	`int16`	`uint16`	`int32`	`uint32`	`int64`	`uint64`	`int`	`uint`	`big.Int`
D	`byte`	`ubyte`	`short`	`ushort`	`int`	`uint`	`long`	`ulong`	N/A	N/A	`BigInt`
Common Lisp[3]											`bignum`
Scheme											
ISLISP[4]											`bignum`
Pascal (FPC)	`shortint`	`byte`	`smallint`	`word`	`longint`	`longword`	`int64`	`qword`	`integer`	`cardinal`	N/A
Visual Basic	N/A	`Byte`	`Integer`	N/A	`Long`	N/A	N/A		N/A		N/A
Visual Basic .NET	`SByte`		`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`			`System.Numerics<wbr/>.BigInteger` (.NET 4.0)

Python 2.x	N/A		N/A		N/A		N/A		int	N/A	long
Python 3.x	N/A		N/A		N/A		N/A		N/A		int
S-Lang	N/A		N/A		N/A		N/A		N/A		N/A
Fortran	INTEGER<wbr/>(KIND = n)[f]	N/A	INTEGER<wbr/>(KIND = n)[f]	N/A	INTEGER<wbr/>(KIND = n)[f]	N/A	INTEGER<wbr/>(KIND = n)[f]	N/A			
PHP	N/A		N/A		int [m]	N/A	int [m]	N/A	N/A		[e]
Perl 5	N/A[d]		N/A[d]		N/A[d]		N/A[d]		N/A[d]		Math::BigInt
Perl 6	int8	uint8	int16	uint16	int32	uint32	int64	uint64	Int	N/A	
Ruby	N/A		N/A		N/A		N/A		Fixnum	N/A	Bignum
Scala	Byte	N/A	Short	Char[l]	Int	N/A	Long	N/A	N/A	N/A	scala.math.BigInt
Seed7	N/A	N/A	N/A	N/A	N/A	N/A	integer	N/A	N/A	N/A	bigInteger
Smalltalk	N/A		N/A		N/A		N/A		SmallInteger[i]	N/A	LargeInteger[i]
Windows PowerShell	N/A		N/A		N/A		N/A		N/A		N/A
OCaml	N/A		N/A		int32	N/A	int64	N/A	int **or** nativeint		open Big_int;; big_int
F#	sbyte	byte	int16	uint16	int32 **or** int	uint32		uint64	nativeint	unativeint	bigint
Standard ML	N/A	Word8.word	N/A		Int32.int	Word32.word	Int64.int	Word64.word	int	word	LargeInt.int **or** IntInf.int
Haskell (GHC)	«import Int» Int8	«import Word» Word8	«import Int» Int16	«import Word» Word16	«import Int» Int32	«import Word» Word32	«import Int» Int64	«import Word» Word64	Int	«import Word» Word	Integer
Eiffel	INTEGER_8	NATURAL_8	INTEGER_16	NATURAL_16	INTEGER_32	NATURAL_32	INTEGER_64	NATURAL_64	INTEGER	NATURAL	N/A
COBOL[h]	BINARY-CHAR «SIGNED»	BINARY-CHAR UNSIGNED	BINARY-SHORT «SIGNED»	BINARY-SHORT UNSIGNED	BINARY-LONG «SIGNED»	BINARY-LONG UNSIGNED	BINARY-DOUBLE «SIGNED»	BINARY-DOUBLE UNSIGNED	N/A	N/A	N/A
Mathematica	N/A		N/A		N/A		N/A		N/A		Integer

^a The standard constants `int shorts` and `int lengths` can be used to determine how many '`short`'s and '`long`'s can be usefully prefixed to '`short int`' and '`long int`'. The actually size of the '`short int`', '`int`' and '`long int`' is available as constants `short max int`, `max int` and `long max int` etc.

^b Commonly used for characters.

^c The ALGOL 68, C and C++ languages do not specify the exact width of the integer types `short`, `int`, `long`, and (C99, C++11) `long long`, so they are implementation-dependent. In C and C++ `short`, `long`, and `long long` types are required to be at least 16, 32, and 64 bits wide, respectively, but can be more. The `int` type is required to be at least as wide as `short` and at most as wide as `long`, and is typically the width of the word size on the processor of the machine (i.e. on a 32-bit machine it is often 32 bits wide; on 64-bit machines it is often 64 bits wide). C99 and C++11Wikipedia:Citation needed also define the `[u]intN_t` exact-width types in the stdint.h header. SeeC syntax#Integral types for more information.

^d Perl 5 does not have distinct types. Integers, floating point numbers, strings, etc. are all considered "scalars".

^e PHP has two arbitrary-precision libraries. The BCMath library just uses strings as datatype. The GMP library uses an internal "resource" type.

^f The value of "n" is provided by the `SELECTED_INT_KIND`[5] intrinsic function.

^g ALGOL 68G's run time option `--precision "number"` can set precision for `long long int`s to the

required "number" significant digits. The standard constants `long long int width` and `long long max int` can be used to determine actual precision.

^h COBOL allows the specification of a required precision and will automatically select an available type capable of representing the specified precision. "`PIC S9999`", for example, would required a signed variable of four decimal digits precision. If specified as a binary field, this would select a 16 bit signed type on most platforms.

^i Smalltalk automatically chooses an appropriate representation for integral numbers. Typically, two representations are present, one for integers fitting the native word size minus any tag bit (SmallInteger) and one supporting arbitrary sized integers (LargeInteger). Arithmetic operations support polymorphic arguments and return the result in the most appropriate compact representation.

^j Ada range types are checked for boundary violations at run-time (as well as at compile-time for static expressions). Run time boundary violations raise a "constraint error" exception. Ranges are not restricted to powers of two. Commonly predefined Integer subtypes are: Positive (`range 1 .. Integer'Last`) and Natural (`range 0 .. Integer'Last`). `Short_Short_Integer` (8 bit), `Short_Integer` (16 bit) and `Long_Integer` (64 bit) are also commonly predefined, but not required by the Ada standard. Run time checks can be disabled if performance is more important than integrity checks.

^k Ada modulo types implement modulo arithmetic in all operations, i.e. no range violations are possible. Modulos are not restricted to powers of two.

^l Commonly used for characters like Java's char.

^m `int` in PHP has the same width as `long` type in C has on that system [c].

Floating point

	Single precision	Double precision	Processor dependent
Ada	`Float`	`Long_Float`	N/A
ALGOL 68	`real`[a]	`long real`[a]	`short real`, `long long real`, etc.[d]
C	`float`[b]	`double`	`long double`[f]
Objective-C			
C++ (STL)			
C#	`float`		N/A
Java			
Go	`float32`	`float64`	
D	`float`	`double`	`real`
Common Lisp			
Scheme			
ISLISP			
Pascal (Free Pascal)	`single`	`double`	`real`
Visual Basic	`Single`	`Double`	N/A
Visual Basic .NET			
Xojo			
Python	N/A	`float`	
JavaScript		`Number`[6]	N/A
S-Lang			
Fortran	`REAL(KIND = n)`[c]		

PHP		`float`	
Perl			
Perl 6	`num32`	`num64`	`Num`
Ruby	N/A	`Float`	N/A
Scala	`Float`	`Double`	
Seed7	N/A	`float`	
Smalltalk	`Float`	`Double`	
Windows PowerShell			
OCaml	N/A	`float`	N/A
F#	`float32`		
Standard ML	N/A	`real`	
Haskell (GHC)	`Float`	`Double`	
Eiffel	`REAL_32`	`REAL_64`	
COBOL	`FLOAT-BINARY-7`[e]	`FLOAT-BINARY-34`[e]	`FLOAT-SHORT`, `FLOAT-LONG`, `FLOAT-EXTENDED`
Mathematica	N/A	N/A	`Real`

^a The standard constants `real shorts` and `real lengths` can be used to determine how many '`short`'s and '`long`'s can be usefully prefixed to '`short real`' and '`long real`'. The actually size of the '`short real`', '`real`' and '`long real`' is available as constants `short max real`, `max real` and `long max real` etc. With the constants `short small real`, `small real` and `long small real` available for each type's machine epsilon.
^b declarations of single precision often are not honored
^c The value of "n" is provided by the `SELECTED_REAL_KIND`[7] intrinsic function.
^d ALGOL 68G's run time option `--precision "number"` can set precision for `long long reals` to the required "number" significant digits. The standard constants `long long real width` and '`long long max real` can be used to determine actual precision.
^e These IEEE floating-point types will be introduced in the next COBOL standard.
^f Same size as '`double`' on many implementations.

Complex numbers

	Integer	Single precision	Double precision	Half and Quadruple precision etc.
Ada	N/A	`Complex`[b]	`Complex`[b]	`Complex`[b]
ALGOL 68	N/A	`compl`	`long compl` etc.	`short compl` etc. & `long long compl` etc.
C (C99) [8]	N/A	`float complex`	`double complex`	N/A
C++ (STL)	N/A	`std::complex<float>`	`std::complex<double>`	
C#	N/A	N/A	`System.Numerics.Complex` (.Net 4.0)	
Java	N/A	N/A	N/A	
Go	N/A	`complex64`	`complex128`	
D	N/A	`cfloat`	`cdouble`	

Objective-C	N/A	N/A	N/A	
Common Lisp				
Scheme				
Pascal	N/A	N/A		
Visual Basic	N/A	N/A		
Visual Basic .NET	N/A	N/A	`System.Numerics.Complex` (.Net 4.0)	
Perl			`Math::Complex`	
Perl 6		`complex64`	`complex128`	`Complex`
Python			`complex`	N/A
JavaScript	N/A	N/A		
S-Lang	N/A	N/A		
Fortran		`COMPLEX(KIND = n)` [a]		
Ruby	`Complex`	N/A	`Complex`	
Scala	N/A	N/A	N/A	
Seed7	N/A	N/A	`complex`	
Smalltalk	`Complex`	`Complex`	`Complex`	
Windows PowerShell	N/A	N/A		
OCaml	N/A	N/A	`Complex.t`	
F#			`System.Numerics.Complex` (.Net 4.0)	
Standard ML	N/A	N/A	N/A	
Haskell (GHC)	N/A	`Complex.Complex Float`	`Complex.Complex Double`	
Eiffel	N/A	N/A	N/A	
COBOL	N/A	N/A	N/A	
Mathematica	`Complex`	N/A	N/A	`Complex`

^a The value of "n" is provided by the `SELECTED_REAL_KIND` intrinsic function.

^b Generic type which can be instantiated with any base floating point type.

Other variable types

	Text		Boolean	Enumeration	Object/Universal
	Character	String[a]			
Ada	`Character`	`String` **`&`** `Bounded_String` **`&`** `Unbounded_String`	`Boolean`	(**item$_1$**, **item$_2$**, ...)	`tagged null record`
ALGOL 68	`char`	`string` **`&`** `bytes`	`bool` **`&`** `bits`	N/A - User defined [9]	N/A
C (C99)	`char` `wchar_t`	N/A	`bool`[b]	**`enum`** «name» **{**item$_1$, item$_2$, ... **};**	`void *`
C++ (STL)		«`std::`»`string`			
Objective-C	`unichar`	`NSString *`	`BOOL`		`id`
C#	`char`	`string`	`bool`	**`enum`** `name` **{**item$_1$, item$_2$, ... **}**	`object`
Java		`String`	`boolean`		`Object`

Go	`rune`	`string`	`bool`	`const (`
				item$_1$ `= iota`**item$_2$**
				...
				`)`
					`interface{}`

D	`char`	`string`	`bool`	`enum` **name** {**item$_1$**, **item$_2$**, ... }	`std.variant.Variant`
Common Lisp					
Scheme					
ISLISP					
Pascal (ISO)	`char`	N/A	`boolean`	(**item$_1$**, **item$_2$**, ...)	N/A
Object Pascal (Delphi)		`string`			`variant`
Visual Basic	N/A	`String`	`Boolean`	`Enum` **name item1 item2 ...** `End Enum`	`Variant`
Visual Basic .NET	`Char`				`Object`
Xojo	N/A				`Object or Variant`
Python	N/A[d]	`str`	`bool`		`object`
JavaScript	N/A[d]	`String`	`Boolean`		`Object`
S-Lang					
Fortran	`CHARACTER(LEN = *)`	`CHARACTER(LEN = :), allocatable`	`LOGICAL(KIND = n)`[f]		`CLASS(*)`
PHP	N/A[d]	`string`	`bool`		`object`
Perl	N/A[d]				
Perl 6	`Char`	`Str`	`Bool`	`enum` **name** <**item$_1$ item$_2$** ...> **or** `enum` **name** <<:**item$_1$**(**value**):**item$_2$**(**value**) ...>>	`Mu`

Ruby	N/A[d]	`String`	`Object`[c]		`Object`
Scala	`Char`	`String`	`Boolean`	`object` **`name`** `extends Enumeration {`**`val item1, item2, ... = Value`**`}`	`Any`
Seed7	`char`	`string`	`boolean`	**`const type:`** `name` **`is new enum`** `item1, item2, ...` **`end enum;`**	
Windows PowerShell					
OCaml	`char`	`string`	`bool`	N/A[e]	N/A
F#				`type` **`name = item1 = value | item2 = value`** `| ...`	`obj`
Standard ML				N/A[e]	N/A
Haskell (GHC)	`Char`	`String`	`Bool`	N/A[e]	N/A
Eiffel	`CHARACTER`	`STRING`	`BOOLEAN`	N/A	`ANY`
COBOL	`PIC X`	`PIC X(`**`string length`**`)` **`or`** `PIC X«X...»`	`PIC 1«(`**`number of digits`**`)»` **`or`** `PIC 1«1...»`	N/A	`OBJECT REFERENCE`
Mathematica	N/A[d]	`String`			N/A

^a specifically, strings of arbitrary length and automatically managed.

^b This language represents a boolean as an integer where false is represented as a value of zero and true by a non-zero value.

^c All values evaluate to either true or false. Everything in **`TrueClass`** evaluates to true and everything in **`FalseClass`** evaluates to false.

^d This language does not have a separate character type. Characters are represented as strings of length 1.

^e Enumerations in this language are algebraic types with only nullary constructors

^f The value of "n" is provided by the `SELECTED_INT_KIND` intrinsic function.

Derived types

Array

Further information: Comparison of programming languages (array)

	fixed size array		dynamic size array	
	one-dimensional array	multi-dimensional array	one-dimensional array	multi-dimensional array
Ada	**array** (<first> .. <last>) **of** <type> or **array** (<discrete_type>) **of** <type>	**array** (<first$_1$> .. <last$_1$>, <first$_2$> .. <last$_2$>, ...) **of** <type> or **array** (<discrete_type$_1$>, <discrete_type$_2$>, ...) **of** <type>	**array** (<discrete_type> **range <>**) **of** <type>	**array** (<discrete_type$_1$> **range <>**, <discrete_type$_2$> **range <>**, ...) **of** <type>
ALGOL 68	[*first*:*last*]**<wbr/>«modename»** or simply: [*size*]**<wbr/>«modename»**	[*first$_1$*:*last$_1$*, *first$_2$*:*last$_2$*]**<wbr/>«modename»** or [*first$_1$*:*last$_1$*][*first$_2$*:*last$_2$*]**<wbr/>«modename»** etc.	**flex**[*first*:*last*]**<wbr/>«modename»** or simply: **flex**[*size*]**<wbr/>«modename»**	**flex**[*first$_1$*:*last$_1$*, *first$_2$*:*last$_2$*]**<wbr/>«modename»** or **flex**[*first$_1$*:*last$_1$*]<wbr/>**flex**[*first$_2$*:*last$_2$*]**<wbr/>«modename»** etc.
C (C99)	type name**[**size**]** [a]	type name**[**size$_1$**][**size$_2$**]** [a]	type ***name or within a block: int n = ...; type name**[**n**]**	
C++ (STL)	**«std::»array<**type, size**>**(C++11)		**«std::»vector<**type**>**	
C#	type**[]**	type**[,,...]**	**System<wbr/>.Collections<wbr/>.ArrayList** or **System<wbr/>.Collections<wbr/>.Generic<wbr/>.List<wbr/><**type**>**	
Java	type**[]** [b]	type**[][]...** [b]	**ArrayList** or **ArrayList<**type**>**	
D	type**[**size**]**	type**[**size$_1$**][**size$_2$**]**	type**[]**	
Go	**[**size**]**type	**[**size$_1$**][**size$_2$**]...**type	**[]type**	**[][]type**
Objective-C	**NSArray**		**NSMutableArray**	
JavaScript	N/A	N/A	**Array** [d]	
Common Lisp				
Scheme				
ISLISP				
Pascal	**array[**first..last**] of** type [c]	**array[**first$_1$..last$_1$**] of array[**first$_2$..last$_2$**] ... of** type [c] or **array[**first$_1$..last$_1$, first$_2$..last$_2$, ...**] of** type [c]	N/A	N/A
Object Pascal (Delphi)			**array of** type	**array of array ... of** type
Visual Basic				
Visual Basic .NET			**System<wbr/>.Collections<wbr/>.ArrayList** or **System<wbr/>.Collections<wbr/>.Generic<wbr/>.List<wbr/>(Of** type**)**	
Python			**list**	
S-Lang				
Fortran	type **::** name**(**size**)**	type **::** name**(**size$_1$, size$_2$,...**)**	type, **ALLOCATABLE ::** name**(:)**	type, **ALLOCATABLE ::** name**(:,:,...)**
PHP			**array**	
Perl				
Perl 6			**Array[**type**]** or **Array of** type	

Ruby			`Array`	
Scala	`Array[type]`	`Array[...[Array[type]]...]`	`ArrayBuffer[type]`	

Seed7	`array type`
	or
	`array [idxType] type`
		`array array type`
		or
		`array [idxType] array [idxType] type`
			`array type`
			or
			`array [idxType] type`
				`array array type`
				or
				`array [idxType] array [idxType] type`

Smalltalk	`Array`		`OrderedCollection`	
Windows PowerShell	`type[]`	`type[,,...]`		
OCaml	`type array`	`type array ... array`		

F#	`type [] or type array`	`type [,,...]`	`System<wbr/>.Collections<wbr/>.ArrayList`
			or
			`System<wbr/>.Collections<wbr/>.Generic<wbr/>.List<wbr/><type>`	

Standard ML	`type vector or type array`			
Haskell (GHC)				
COBOL	`level-number type OCCURS size «TIMES».`	`one-dimensional array definition...`	`level-number type OCCURS min-size TO max-size «TIMES» DEPENDING «ON» size.`[e]	N/A

^a In most expressions (except the `sizeof` and `&` operators), values of array types in C are automatically converted to a pointer of its first argument. See C syntax#Arrays for further details of syntax and pointer operations.

^b The C-like "type `x[]`" works in Java, however "type `[] x`" is the preferred form of array declaration.

^c Subranges are used to define the bounds of the array.

^d JavaScript's array are a special kind of object.

^e The `DEPENDING ON` clause in COBOL does not create a 'true' variable length array and will always allocate the maximum size of the array.

Other types

	Simple composite types		Algebraic data types	Unions
	Records	Tuple expression		

	type identifier **is** «**abstract**» «**tagged**» «**limited**» [**record** fieldname$_1$: type; fieldname$_2$: type; ... **end record**	**null record**]	N/A	Any combination of records, unions and enumerations (as well as references to those, enabling recursive types).	**type** identifier (variation : discrete_type) **is r** **case** variation **when** choice_li **=>** fiel : ty ... **when** choice_li **=>** fiel : ty ... ... **end case;** **end record**
68	**struct** (**modename** «*fieldname*», ...);	Required types and operators can be user defined			**union** (**modename**, ..
e-C	**struct** «name» {type name;...};	N/A	N/A		**union** {type name;..
	struct «name» {type name;...};[b]	«std::»tuple<type$_1$..type$_n$>			
	struct name {type name;...}				N/A
	N/A[a]				
pt		N/A			
	struct name {type name;...}		**std.variant.Algebraic**!(type,...)		**union** {type name;..
	struct {**«name» type...**}				
n		(**cons** val$_1$ val$_2$)[c]			
	N/A				
	record name: type;... **end**	N/A	N/A		**record** **case** type **of** value: (types) **end**
ET	**Structure** name **Dim** name **As** type ... **End Structure**				
	N/A[a]	«(»val$_1$, val$_2$, val$_3$, ... «)»			N/A
	struct {name [=value], ...}				

	TYPE name type **::** name ... **END TYPE**			
	N/A[a]			
	N/A[d]			N/A
	N/A[a]			
	`<font size="8.10">OpenStruct.new({:name => value})</font>`			

	`case class name(«var» name: type, ...)`	`(val1, val2, val3, ... )`	`abstract class name`
			`case class Foo(«parameters») extends name`
			`case class Bar(«parameters») extends name`
			`...`
			or
			`abstract class name`
			`case object Foo extends name`
			`case object Bar extends name`
			`...`
			or combination of case classes and case objects	

s hell				
	`type name = {«mutable» name : type;...}`	`«(»val1, val2, val3, ... «)»`	`type name = Foo «of type» | Bar «of type» | ...`	N/A
l	`type name = {name : type,...}`	`(val1, val2, val3, ... )`	`datatype name = Foo «of type» | Bar «of type» | ...`	
	`data Name = Constr {name :: type,...}`		`data Name = Foo «types» | Bar «types» | ...`	

	`level-number name type clauses.`
	`level-number+n name type clauses.`
	`...`
		N/A	N/A	`name REDEFINES vari`
				`type.`

^a Only classes are supported.

^b `structs` in C++ are actually classes, but have default public visibility and *are* also POD objects. C++11 extended this further, to make classes act identically to POD objects in many more cases.

^c pair only

^d Although Perl doesn't have records, because Perl's type system allows different data types to be in an array, "hashes" (associative arrays) that don't have a variable index would effectively be the same as records.

^e Enumerations in this language are algebraic types with only nullary constructors

Variable and constant declarations

	variable	constant	type synonym
Ada	`identifier : type «:= initial_value»`[e]	`identifier : constant type := final_value`	`subtype identifier is type`
ALGOL 68	**modename** *name* «:= initial_value»;	**modename** *name* = value;	**mode synonym** = **modename**;
C (C99)	`type name «= initial_value»;`	`enum{ name = value };`	`typedef type synonym;`
Objective-C			
C++		`const type name = value;`	
C#	`type name «= initial_value»;` or `var name = value;`	`const type name = value;` or `readonly type name = value;`	`using synonym = type;`
D	`type name «= initial_value»;` or `auto name = value;`	`const type name = value;` or `immutable type name = value;`	`alias type synonym;`
Java	`type name «= initial_value»;`	`final type name = value;`	N/A
JavaScript	`var name «= initial_value»;`	`const name = value;`	
Go	`var name type «= initial_value»` or `name := initial_value`	`const name «type» = initial_value`	`type synonym type`
Common Lisp	`(defparameter name initial_value)` or `(defvar name initial_value)` or `(setf (symbol-value 'symbol) initial_value)`	`(defconstant name value)`	`(deftype synonym () 'type)`
Scheme	`(define name initial_value)`		
ISLISP	`(defglobal name initial_value)` or `(defdynamic name initial_value)`	`(defconstant name value)`	N/A
Pascal[a]	`name: type «= initial_value»`	`name = value`	`synonym = type`
Visual Basic	`Dim name As type`	`Const name As type = value`	
Visual Basic .NET	`Dim name As type«= initial_value»`		`Imports synonym = type`
Xojo	`Dim name As type«= initial_value»`		N/A
Python	`name = initial_value`	N/A	`synonym = type`[b]
CoffeeScript			N/A
S-Lang	`name = initial_value;`		`typedef struct {...} typename`
Fortran	type name	`type, PARAMETER :: name = value`	
PHP	`$name = initial_value;`	`define("name", value);` `const name = value` (5.3+)	N/A
Perl	«**my**» `$name «= initial_value»;`[c]	`use constant name => value;`	

Perl 6	«**my** «type»» $name «**=** initial_value»;[c]	«**my** «type»» **constant** name **=** value;	::synonym ::= type
Ruby	name **=** initial_value	Name **=** value	synonym **=** type[b]
Scala	**var** name«: type» **=** initial_value	**val** name«: type» **=** value	**type** synonym **=** type
Windows PowerShell	«**[**type**]**» **$**name **=** initial_value	N/A	N/A
Bash shell	name**=**initial_value	N/A	N/A
OCaml	**let** name «: type **ref**» **= ref** value[d]	**let** name «: type» **=** value	**type** synonym **=** type
F#	**let mutable** name «: type» **=** value		
Standard ML	**val** name «: type **ref**» **= ref** value[d]	**val** name «: type» **=** value	
Haskell		«name::type;» name **=** value	**type** Synonym **=** type
Forth	**VARIABLE** name (in some systems use value **VARIABLE** name instead)	value **CONSTANT** name	
COBOL	level-number name type clauses.	«**0**»**1** name **CONSTANT** «**AS**» value.	level-number name type clauses «**IS**» **TYPEDEF.**
Mathematica	name**=**initialvalue	N/A	N/A

^a Pascal has declaration blocks. See Comparison of programming languages (basic instructions)#Functions.

^b Types are just regular objects, so you can just assign them.

^c In Perl, the "my" keyword scopes the variable into the block.

^d Technically, this does not declare *name* to be a mutable variable—in ML, all names can only be bound once; rather, it declares *name* to point to a "reference" data structure, which is a simple mutable cell. The data structure can then be read and written to using the `!` and `:=` operators, respectively.

^[e] If no initial value is given, an invalid value is automatically assigned (which will trigger a run-time exception if it used before a valid value has been assigned). While this behaviour can be suppressed it is recommended in the interest of predictability. If no invalid value can be found for a type (for example in case of an unconstraint integer type), a valid, yet predictable value is chosen instead.

Control flow

Conditional statements

	if	else if	select case	conditional expression

Ada	**if** condition **then**
	statements
	«**else**
	statements»
	end if
		if condition$_1$ **then**
		statements
		elsif condition$_2$ **then**
		statements
		...
		«**else**
		statements»
		end if
			case expression **is**
			when value_list$_1$ **=>** statements
			when value_list$_2$ **=>** statements
			...
			«**when others =>** statements»
			end case
				(**if** condition$_1$ **then**
				expression$_1$
				«**elsif** condition$_2$ **then**
				expression$_2$»
				...
				else
				expression$_n$
				)
				(**case** expression **is**
				when value_list$_1$ **=>** expression$_1$
				when value_list$_2$ **=>** expression$_2$
				...
				«**when others =>** expression$_n$»
				)

Seed7			**case** expression **of** **when** set1 **:** statements ... «**otherwise:** statements» **end case**	
Modula-2	**if** condition **then** statements «**else** statements» **end**	**if** condition$_1$ **then** statements **elsif** condition$_2$ **then** statements ... «**else** statements» **end**	**case** expression **of** caseLabelList **:** statements **|** ... «**else** statements» **end**	
ALGOL 68 & "brief form"	**if** condition **then** statements «**else** statements» **fi** (condition | statements «| statements»)	**if** condition **then** statements **elif** condition **then** statements **fi** (condition | statements |: condition | statements)	**case** switch **in** statements, statements«,... **out** statements» **esac** (variable | statements,... «| statements»)	(condition | valueIfTrue | valueIfFalse)
C (C99) Objective-C C++ (STL) D Java JavaScript PHP	**if** (condition) {instructions} «**else** {instructions}»	**if** (condition) {instructions} **else if** (condition) {instructions} ... «**else** {instructions}»	**switch** (variable) {**case case1: instructions «break;»...«default: instructions»**} **switch** (variable) {**case case1: instructions; «jump statement;»...«default: instructions; «jump statement;»»** }	condition **?** valueIfTrue **:** valueIfFalse
C#				
Windows PowerShell		**if** (condition) { instructions } **elseif** (condition) { instructions } ... «**else** { instructions }»	**switch** (variable) { case1 { instructions «**break;**» } ... «**default** { instructions }»}	

Go	**if** condition **{**instructions**}** «**else** **{**instructions**}**»	**if** condition **{**instructions**}** **else if** condition **{**instructions**}** ... «**else** **{**instructions**}**» or **switch** **{case condition: instructions ...«default: instructions»**}	**switch** variable **{case case1: instructions ...«default: instructions»**}	
Perl	**if** **(**condition**)** **{**instructions**}** «**else** **{**instructions**}**» or **unless** **(**notcondition**)** **{**instructions**}** «**else** **{**instructions**}**»	**if** **(**condition**)** **{**instructions**}** **elsif** **(**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**» or **unless** **(**notcondition**)** **{**instructions**}** **elsif** **(**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**»	**use feature "switch";** ... **given** **(**variable**)** **{when (case1) { instructions }...«default { instructions }»**}	condition **?** valueIfTrue **:** valueIfFalse
Perl 6	**if** condition **{**instructions**}** «**else** **{**instructions**}**» or **unless** notcondition **{**instructions**}**	**if** condition **{**instructions**}** **elsif** condition **{**instructions**}** ... «**else** **{**instructions**}**	**given** variable **{when case1 { instructions }...«default { instructions }»**}	condition **??** valueIfTrue **!!** valueIfFalse
Ruby	**if** condition instructions «**else** instructions»	**if** condition instructions **elsif** condition instructions ... «**else** instructions» **end**	**case** variable **when** case1 instructions ... «**else** instructions» **end**	condition **?** valueIfTrue **:** valueIfFalse
Scala	**if** **(**condition**)** **{**instructions**}** «**else** **{**instructions**}**»	**if** **(**condition**)** **{**instructions**}** **else if** **(**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**»	expression **match** **{** **case** pattern1 **=>** expression **case** pattern2 **=>** expression ... «**case** **_** **=>** expression» **}**[b]	**if** **(**condition**)** valueIfTrue **else** valueIfFalse
Smalltalk	condition **ifTrue:** trueBlock «**ifFalse:** falseBlock» **end**			condition **ifTrue:** trueBlock **ifFalse:** falseBlock
Common Lisp	**(when** condition instructions**)** or **(unless** condition instructions**)** or **(if** condition**(progn instructions)«(progn instructions)»)**	**(cond** **(**condition1 instructions**)** **(condition2 instructions)...«(t instructions)»)**	**(case** expression**(case1 instructions)(case2 instructions)...«(otherwise instructions)»)**	**(if** condition valueIfTrue valueIfFalse**)**
Scheme	**(when** conditioninstructions**)** or **(if** condition **(begin** instructions**)** «**(begin** instructions**)**»)	**(cond** **(**condition1 instructions**)** **(**condition2 instructions**)** ...«**(else** instructions**)**»)	**(case** **(**variable**)** **((**case1**)** instructions**)** **((**case2**)** instructions**)** ...«**(else** instructions**)**»)	

ISLISP	**(if** condition**(progn instructions)«(progn instructions)»)**	**(cond (**condition1 instructions**)(condition2 instructions)...«(t instructions)»)**	**(case** expression**(case1 instructions)(case2 instructions)...«(t instructions)»)**	**(if** condition valueIfTrue valueIfFalse**)**
Pascal	**if** condition **then begin** instructions **end** «**else begin** instructions **end**»[c]	**if** condition **then begin** instructions **end** **else if** condition **then begin** instructions **end** ... «**else begin** instructions **end**»[c]	**case** variable **of** case1: instructions ... «**else**: instructions» **end**[c]	

Visual Basic	**If** condition **Then**
	instructions
	«**Else**
	instructions»
	End If
		If condition **Then**
		instructions
		ElseIf condition **Then**
		instructions
		...
		«**Else**
		instructions»
		End If
			Select Case variable
			Case case1
			instructions
			...
			«**Case Else**
			instructions»
			End Select
				IIf(condition, valueIfTrue, valueIfFalse**)**

Visual Basic .NET				**If(**condition, valueIfTrue, valueIfFalse**)**
Xojo				
Python [a]	**if** condition **:** Tab ▯ instructions «**else:** Tab ▯ instructions»	**if** condition **:** Tab ▯ instructions **elif** condition **:** Tab ▯ instructions ... «**else:** Tab ▯ instructions»		valueIfTrue **if** condition **else** valueIfFalse (Python 2.5+)
S-Lang	**if (**condition**) { }** instructions **«else {** instructions **}»**	**if (**condition**) { }** instructions **else if (**condition**) { }** instructions **... «else { }** instructions **»**	**switch (**variable**) { case** case1: instructions **} { case** case2: instructions **}** ...	
Fortran	**IF (**condition**) THEN** instructions **ELSE** instructions **ENDIF**	**IF (**condition**) THEN** instructions **ELSEIF (**condition**) THEN** instructions . . . **ELSE** instructions **ENDIF**	**SELECT CASE (**variable**) CASE (case1)** instructions ... **CASE DEFAULT** instructions **END SELECT**	
Forth	condition **IF** instructions **«** **ELSE** instructions**»** **THEN**	condition **IF** instructions **ELSE** condition **IF** instructions **THEN THEN**	value **CASE** case **OF** instructions **ENDOF** case **OF** instructions **ENDOF** default instructions **ENDCASE**	condition **IF** valueIfTrue **ELSE** valueIfFalse **THEN**

OCaml	**if** condition **then begin** instructions **end** «**else begin** instructions **end**»	**if** condition **then begin** instructions **end else if** condition **then begin** instructions **end** ... «**else begin** instructions **end**»	**match** value **with pattern1 -> expression| pattern2 -> expression ... «| _ -> expression»#endnote_pattern matching[b]**	**if** condition **then** valueIfTrue **else** valueIfFalse
F#	**if** condition **then** Tab ▯ instructions «**else** Tab ▯ instructions»	**if** condition **then** Tab ▯ instructions **elif** condition **then** Tab ▯ instructions ... «**else** Tab ▯ instructions»		
Standard ML	**if** condition **then** «**(**»instructions «**)**» **else** «**(**» instructions «**)**»	**if** condition **then** «**(**»instructions «**)**» **else if** condition **then** «**(**» instructions «**)**» ... **else** «**(**» instructions «**)**»	**case** value **ofpattern1 => expression | pattern2 => expression ... «| _ => expression»#endnote_pattern matching[b]**	
Haskell (GHC)	**if** condition **then** expression **else** expression or **when** condition **(do** instructions**)** or **unless** notcondition **(do** instructions**)**	result **|** condition **=** expression **|** condition **=** expression **| otherwise =** expression	**case** value **of {pattern1 -> expression;pattern2 ->expression; ... «_ -> expression»}** [b]	
Bash shell	**if** condition-command; **then** expression «**else** expression» **fi**	**if** condition-command; **then** expression **elif** condition-command; **then** expression «**else** expression» **fi**	**case** "$variable" **in** "$condition1") command... "$condition2") command... **esac**	
CoffeeScript	**if** condition **then** expression «**else** expression»	**if** condition **then** expression **else if** condition **then** expression «**else** expression»	**switch** expression **when** condition **then** expression **else** expression	*All conditions are expressions*
	if condition expression «**else** expression»	**if** condition expression **else if** condition expression «**else** expression»		
	expression **if** condition	**unless** condition expression **else unless** condition expression «**else** expression»	**switch** expression **when** condition expression «**else** expression»	
	unless condition expression «**else** expression»			
	expression **unless** condition			
COBOL	**IF** condition «**THEN**» expression «**ELSE** expression». [d]		**EVALUATE** expression «**ALSO** expression...» **WHEN** case-or-condition «**ALSO** case-or-condition...» expression ... «**WHEN OTHER** expression» **END-EVALUATE**	

	if	else if	select case	conditional expression

^a A single instruction can be written on the same line following the colon. Multiple instructions are grouped together in a block which starts on a newline (The indentation is required). The conditional expression syntax does not follow this rule.

^b This is pattern matching and is similar to select case but not the same. It is usually used to deconstruct algebraic data types.

^c In languages of the Pascal family, the semicolon is not part of the statement. It is a separator between statements, not a terminator.

^d **`END-IF`** may be used instead of the period at the end.

Loop statements

	while	do while	for i = first to last	foreach
Ada	**while** condition **loop** statements **end loop**	**loop** statements **exit when not** condition **end loop**	**for** index **in** «**reverse**» [first .. last | discrete_type] **loop** statements **end loop**	**for** item **of** «**reverse**» iterator **loop** statements **end loop** **(for** [**all** | **some**] [**in** | **of**] [first .. last | discrete_type | iterator] **=>** predicate**)** [b]
ALGOL 68	«**for** index» «**from** first» «**by** increment» «**to** last» «**while** condition» **do** statements **od**			**for** *key* «**to** **upb** *list*» **do** «**typename** *val=list*[*key*]**;**» statements **od**
	«**while** condition» **do** statements **od**	«**while** statements**;** condition» **do** statements **od**	«**for** index» «**from** first» «**by** increment» «**to** last» **do** statements **od**	

C (C99)	`while (condition) { instructions }`	`do { instructions } while (condition)`	`for («type» i = first; i <= last; ++i) { instructions }`	N/A
Objective-C				`for (type item in set) { instructions }`
C++ (STL)				«`std::»for_each(start, end, function)` (C++11) `for (type item : set) { instructions }`
C#				`foreach (type item in set) { instructions }`
Java				`for (type item : set) { instructions }`

JavaScript			`for (var i = first; i <= last; i++) { instructions }`	`for (var index in set) { instructions }`
				or
				`for each (var item in set) { instructions }`
				(JS 1.6+, deprecated[10])
				or
				`for (var item of set) { instructions }`
				(EcmaScript 6 proposal, supported in Firefox[11])

PHP			`foreach (range(first, last-1) as $i) { instructions }` or `for ($i = first; $i <= last; $i++) { instructions }`	`foreach (set as item) { instructions }`
				or
				`foreach (set as key => item) { instructions }`

Windows PowerShell			`for ($i = first; $i -le last; $i++) { instructions }`	`foreach (item in set) { instructions using item }`
D			`foreach (i; first ... last) { instructions }`	`foreach («type» item; set) { instructions }`
Go	`for condition { instructions }`		`for i := first; i <= last; i++ { instructions }`	`for key, item := range set { instructions }`
Perl	`while (condition) { instructions }` or `until (notcondition) { instructions }`	`do { instructions } while (condition)` or `do { instructions } until (notcondition)`	`for«each»«$i» (0 .. N-1) { instructions }` or `for ($i = first; $i <= last; $i++) { instructions }`	`for«each» «$item» (set) { instructions }`
Perl 6	`while condition { instructions }` or `until notcondition { instructions }`	`repeat { instructions } while condition` or `repeat { instructions } until notcondition`	`for first..last -> $i { instructions }` or `loop ($i = first; $i <=last; $i++) { instructions }`	`for set« -> $item» { instructions }`

Ruby	**while** condition instructions **end** or **until** notcondition instructions **end**	**begin** instructions **end while** condition or **begin** instructions **end until** notcondition	**for i in** first...last instructions **end** or first.**upto**(last**-1**) **{** **|i|** instructions **}**	**for** item **in** set instructions **end** or set.**each { |**item**|** instructions **}**
Bash shell	**while** condition **;do** instructions **done** or **until** notcondition **;do** instructions **done**	N/A	**for ((**i **=** first; i **<=** last; **++**i**)) ; do** instructions **done**	**for** item **in** set **;do** instructions **done**
Scala	**while (**condition**) {** instructions **}**	**do {** instructions **}** **while (**condition**)**	**for (**i **<-** first **to** last «**by 1**») **{** instructions **}** or first **to** last «**by 1**» **foreach (**i **=> {** instructions **})**	**for (**item **<-** set**) {** instructions **}** or set **foreach (**item **=> {** instructions **})**
Smalltalk	conditionBlock **whileTrue:** loopBlock	loopBlock **doWhile:** conditionBlock	first **to:** last **do:** loopBlock	collection **do:** loopBlock
Common Lisp	**(loopwhile conditiondoinstructions) or** (do () (**notcondition**) **instructions**)	**(loopdoinstructionswhile condition)**	**(loopfor i from first to last «by 1»doinstructions) or** (dotimes (i N) **instructions) or** (do ((i **first** (1+ i))) ((>= i **last**)) **instructions**)	**(loopfor item in setdoinstructions) or** (dolist (**item set**) **instructions**) or **(mapc** function list**)** or **(map** 'type function sequence**)**
Scheme	**(do () (**notcondition**)** instructions**)** or **(let loop () (if** condition **(begin** instructions **(loop))))**	**(let loop ()** **(**instructions **(if** condition **(loop))))**	**(do ((i** first **(+ i** **1))) ((>= i** last**))** instructions**)** or **(let loop ((i** first**)) (if (< i** last**) (begin** instructions **(loop** **(+ i 1)))))**	**(for-each (lambda** **(**item**)** instructions**)** list**)**
ISLISP	**(while** conditioninstructions**)**	**(tagbody loop** instructions **(if** condition **(go** **loop))**	**(for ((i** first **(+ i** **1))) ((>= i** last**))** instructions**)**	**(mapc (lambda (**item**)** instructions**)** list**)**

Pascal	**while** condition **do**
	begin
	instructions
	end
		repeat
		instructions
		until notcondition;
			for i **:=** first «**step**
			1» **to** last **do begin**
			instructions
			end;[a]
				for item **in** set **do** ...

Visual Basic	**Do While** condition instructions **Loop** or **Do Until** notcondition instructions **Loop**	**Do** instructions **Loop While** condition or **Do** instructions **Loop Until** notcondition	**For** i **=** first **To** last «**Step 1**» instructions **Next i**	**For Each** item **In** set instructions **Next** item
Visual Basic .NET			**For i** «**As** type» **=** first **To** last «**Step 1**» instructions **Next i**[a]	**For Each** item **As** type **In** set instructions **Next** item
Xojo	**While** condition instructions **Wend**	**Do Until** notcondition instructions **Loop** or **Do** instructions **Loop Until** notcondition		
Python	**while** condition **:** Tab instructions «**else:** Tab instructions»	N/A	**for i in range(**first, last**):** Tab instructions «**else:** Tab instructions»(Python 3.x) **for i in xrange(**first, last**):** Tab instructions «**else:** Tab instructions»(Python 2.x)	**for** item **in** set**:** Tab instructions «**else:** Tab instructions»
S-Lang	**while (**condition**) {** instructions **}** «**then** optional-block»	**do {** instructions **}** **while (**condition**)** «**then** optional-block»	**for (i =** first**; i <** last**; i++) {** instructions **}** «**then** optional-block»	**foreach** item**(**set**)** «**using (**what**)**» **{** instructions **}** «**then** optional-block»
Fortran	**DO WHILE (**condition**)** **instructions ENDDO**	**DO instructions IF (condition) EXIT ENDDO**	**DO** I **=** first**,**last **instructions ENDDO**	N/A
Forth	**BEGIN** « instructions » condition **WHILE** instructions **REPEAT**	**BEGIN** instructions condition **UNTIL**	limit start **DO** instructions **LOOP**	N/A
OCaml	**while** condition **do** instructions **done**	N/A	**for i =** first **to** last**−1 do** instructions **done**	**Array.iter (fun** item **->** instructions**)** array **List.iter (fun** item **->** instructions**)** list
F#	**while** condition **do** Tab instructions	N/A	**for i =** first **to** last**−1 do** Tab instructions	**for** item **in** set **do** Tab instructions or **Seq.iter (fun** item **->** instructions**)** set

Standard ML	`while condition do ( instructions )`	N/A		`Array.app (fn item => instructions) array app (fn item => instructions) list`
Haskell (GHC)	N/A		`Control.Monad.forM_ [0..N-1] (\i -> do instructions)`	`Control.Monad.forM_ list (\item -> do instructions)`

Eiffel	`from`
	`  setup`
	`until`
	`  condition`
	`loop`
	`  instructions`
	`end`			

CoffeeScript	`while condition` `  expression`	N/A	`for i in [first..last]` `  expression`	`for item in set` `  expression`
	`expression while condition`			
	`while condition then expression`			
	`until condition` `  expression`			
	`expression until condition`		`for i in [first..last] then expression`	`for item in set then expression`
	`until condition then expression`		`expression for i in [first..last]`	`expression for item in set`
COBOL	`PERFORM procedure-1 «THROUGH procedure-2» ««WITH» TEST BEFORE» UNTIL condition`[c]	`PERFORM procedure-1 «THROUGH procedure-2» «WITH» TEST AFTER UNTIL condition`[c]	`PERFORM procedure-1 «THROUGH procedure-2» VARYING i FROM first BY increment UNTIL i > last`[d]	N/A

	`PERFORM ««WITH» TEST BEFORE» UNTIL condition`
	`  expression`
	`END-PERFORM`
		`PERFORM «WITH» TEST AFTER UNTIL condition`
		`  expression`
		`END-PERFORM`
			`PERFORM VARYING i FROM first BY increment UNTIL i > last`
			`  expression`
			`END-PERFORM`[d]	

^a "`step` n" is used to change the loop interval. If "`step`" is omitted, then the loop interval is 1.

^b This implements the universal quantifier ("for all" or "∀") as well as the existential quantifier ("there exists" or "∃").

^c `THRU` may be used instead of `THROUGH`.

^d «`IS`» `GREATER` «`THAN`» may be used instead of `>`.

Exceptions

Further information: Exception handling syntax

	throw	handler	assertion

Ada	`raise exception_name «with string_expression»`	`begin`
		`  statements`
		`exception`
		`  when exception_list1 => statements;`
		`  when exception_list2 => statements;`
		`...`
		`  «when others => statements;»`
		`end`[b]
			`pragma Assert («Check =>» boolean_expression ««Message =>» string_expression»)`
			`[function | procedure | entry] with`
			`  Pre => boolean_expression`
			`  Post => boolean_expression`
			
			`any_type with Type_Invariant => boolean_expression`

C (C99)	`longjmp(state, exception);`	`switch (setjmp(state)) { case 0: instructions break; case exception: instructions ... }`	`assert(condition);`
C++	`throw exception;`	`try { instructions } catch «(exception)» { instructions } ...`	
C#		`try { instructions } catch «(exception)» { instructions } ... «finally { instructions }»`	`Debug.Assert(condition);`
Java		`try { instructions } catch (exception) { instructions } ... «finally { instructions }»`	`assert condition;`
JavaScript		`try { instructions } catch (exception) { instructions } «finally { instructions }»`	*?*
D		`try { instructions } catch (exception) { instructions } ... «finally { instructions }»`	`assert(condition);`
PHP		`try { instructions } catch (exception) { instructions } «finally { instructions }»`	`assert(condition);`
S-Lang		`try { instructions } catch «exception» { instructions } ... «finally { instructions }»`	*?*
Windows PowerShell		`trap «[exception]» { instructions } ... instructions` or `try { instructions } catch «[exception]» { instructions } ... «finally { instructions }»`	`[Debug]::Assert(condition)`
Objective-C	`@throw exception;`	`@try { instructions } @catch (exception) { instructions } ... «@finally { instructions }»`	`NSAssert(condition, description);`
Perl	`die exception;`	`eval { instructions }; if ($@) { instructions }`	*?*
Perl 6		`try { instructions CATCH { when exception { instructions } ...}}`	*?*

Ruby	**raise** exception	**begin** instructions **rescue** exception instructions ... «**else** instructions» «**ensure** instructions» **end**	
Smalltalk	exception **raise**	instructionBlock **on:** exception **do:** handlerBlock	**assert:** conditionBlock
Common Lisp	**(error** "exception"**)** or **(error (make-conditiontypearguments))**	**(handler-case(progn instructions)(exception instructions)...) or** (handler-bind (**condition (lambdainstructions«invoke-restart restart args»))**...) [a]	**(assert** condition**)** or **(assert** condition «**(**place**)** «error»»**)** or **(check-type** var type**)**
Scheme (R^6RS)	**(raise** exception**)**	**(guard (con (**conditioninstructions**) ...)** instructions**)**	*?*
ISLISP	**(error** "error-string" objects**)** or **(signal-condition** condition continuable**)**	**(with-handlerhandler form***)	*?*
Pascal	**raise** Exception.Create()	**try** Except **on** E: exception **do begin** instructions **end; end;**	*?*

Visual Basic	**Err.Raise** ERRORNUMBER	**With New** Try: **On Error Resume Next**
		OneInstruction
		.Catch: **On Error GoTo 0: Select Case** .Number
		Case ERRORNUMBER
		instructions
		End Select: End With
		`'*** Try class ***`
		`Private mstrDescription As String`
		`Private mlngNumber As Long`
		`Public Sub Catch()`
		`mstrDescription = Err.Description`
		`mlngNumber = Err.Number`
		`End Sub`
		`Public Property Get Number() As Long`
		`Number = mlngNumber`
		`End Property`
		`Public Property Get Description() As String`
		`Description = mstrDescription`
		`End Property`
		[12]
			Debug.Assert condition

Visual Basic .NET	**Throw** exception	**Try** instructions **Catch** «exception» «**When** condition» instructions ... «**Finally** instructions» **End Try**	**Debug.Assert(**condition**)**
Xojo	**Raise** exception	**Try** instructions **Catch** «exception» instructions ... «**Finally** instructions» **End Try**	N/A

Python	**raise** exception	**try:** Tab instructions **except** «exception»: Tab instructions ... «**else:** Tab instructions» «**finally:** Tab instructions»	**assert** condition
Fortran	N/A		
Forth	code **THROW**	xt **CATCH** (code or 0)	N/A
OCaml	**raise** exception	**try** expression **with** pattern **->** expression ...	**assert** condition
F#		**try** expression **with** pattern **->** expression ... or **try** expression **finally** expression	
Standard ML	**raise** exception «arg»	expression **handle** pattern **=>** expression ...	
Haskell (GHC)	**throw** exception *or* **throwError** expression	**catch** tryExpression catchExpression *or* **catchError** tryExpression catchExpression	**assert** condition expression
COBOL	**RAISE** «**EXCEPTION**» exception	**USE** «**AFTER**» **EXCEPTION OBJECT** class-name. or **USE** «**AFTER**» **EO** class-name. or **USE** «**AFTER**» **EXCEPTION CONDITION** exception-name «**FILE** file-name». or **USE** «**AFTER**» **EC** exception-name «**FILE** file-name».	N/A

^a Common Lisp allows `with-simple-restart`, `restart-case` and `restart-bind` to define restarts for use with `invoke-restart`. Unhandled conditions may cause the implementation to show a restarts menu to the user before unwinding the stack.

^b Uncaught exceptions are propagated to the innermost dynamically enclosing execution. Exceptions are not propagated across tasks (unless these tasks are currently synchronised in a rendezvous).

Other control flow statements

	exit block(break)	continue	label	branch (goto)	return value from generator
Ada	**exit** «loop_name» «**when** condition»	N/A	label:	**goto** label	N/A
ALGOL 68	value **exit**; ...	**do** statements; **skip** **exit**; label: statements **od**	label: ...	**go to** label; ... **goto** label; ... label; ...	*yield*(*value*) (Callback)[13]

<table>
<tr><td>C (C99)</td><td rowspan="5">break;</td><td rowspan="5">continue;</td><td rowspan="5">label:</td><td rowspan="5">goto label;</td><td rowspan="4">N/A</td></tr>
<tr><td>Objective-C</td></tr>
<tr><td>C++ (STL)</td></tr>
<tr><td>D</td></tr>
<tr><td>C#</td><td>yield return value;</td></tr>
<tr><td>Java</td><td rowspan="2">break «label»;</td><td rowspan="2">continue «label»;</td><td rowspan="6"></td><td rowspan="2">N/A</td><td></td></tr>
<tr><td>JavaScript</td><td>yield value«;»</td></tr>
<tr><td>PHP</td><td>break «levels»;</td><td>continue «levels»;</td><td>goto label;</td><td>yield «key =>» value;</td></tr>
<tr><td>Perl</td><td rowspan="2">last «label»;</td><td rowspan="2">next «label»;</td><td rowspan="3"></td><td></td></tr>
<tr><td>Perl 6</td><td></td></tr>
<tr><td>Go</td><td>break «label»</td><td>continue «label»</td><td>goto label</td><td></td></tr>
<tr><td>Bash shell</td><td>break «levels»</td><td>continue «levels»</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>Common Lisp</td><td>(return) or (return-from block) or (loop-finish)</td><td></td><td>(tagbody tag ... tag ...)</td><td>(go tag)</td><td></td></tr>
<tr><td>Scheme</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>ISLISP</td><td>(return-from block)</td><td></td><td>(tagbody tag ... tag ...)</td><td>(go tag)</td><td></td></tr>
<tr><td>Pascal(ISO)</td><td colspan="2">N/A</td><td rowspan="2">label:[a]</td><td rowspan="2">goto label;</td><td rowspan="5">N/A</td></tr>
<tr><td>Pascal(FPC)</td><td>break;</td><td>continue;</td></tr>
<tr><td>Visual Basic</td><td rowspan="3">Exit block</td><td>N/A</td><td>label:</td><td rowspan="3">GoTo label</td></tr>
<tr><td>Visual Basic .NET</td><td rowspan="2">Continue block</td><td rowspan="2"></td></tr>
<tr><td>Xojo</td></tr>
<tr><td>Python</td><td>break</td><td>continue</td><td colspan="2">N/A</td><td>yield value</td></tr>
<tr><td>RPG IV</td><td>LEAVE;</td><td>ITER;</td><td></td><td></td><td></td></tr>
<tr><td>S-Lang</td><td>break;</td><td>continue;</td><td></td><td></td><td></td></tr>
<tr><td>Fortran</td><td>EXIT</td><td>CYCLE</td><td>label[b]</td><td>GOTO label</td><td>N/A</td></tr>
<tr><td>Ruby</td><td>break</td><td>next</td><td></td><td></td><td></td></tr>
<tr><td>Windows PowerShell</td><td>break «label»</td><td>continue</td><td></td><td></td><td></td></tr>
<tr><td>OCaml</td><td colspan="4" rowspan="4">N/A</td><td></td></tr>
<tr><td>F#</td><td></td></tr>
<tr><td>Standard ML</td><td></td></tr>
<tr><td>Haskell (GHC)</td><td></td></tr>
</table>

COBOL	**EXIT PERFORM** or **EXIT PARAGRAPH** or **EXIT SECTION** or **EXIT.**	**EXIT PERFORM CYCLE**	label «**SECTION**».	**GO TO** label	N/A
Ya	**break** «from where»; f.e. **break for switch;**	**continue** «to where»; f.e. **continue for switch;**	:label	**goto** :label;	N/A

^a Pascal has declaration blocks. See Comparison of programming languages (basic instructions)#Functions.

^b label must be a number between 1 and 99999.

Functions

See reflection for calling and declaring functions by strings.

	calling a function	basic/void function	value-returning function	required main function
Ada	*foo* «(parameters)»	**procedure** *foo* «(parameters)» **is begin** statements **end** foo	**function** *foo* «(parameters)» **return** type **is begin** statements **end** foo	N/A
ALGOL 68	*foo*«(parameters)»;	**proc** *foo* = «(parameters)» void: (*instructions*);	**proc** *foo* = «(parameters)» **rettype**: (*instructions ...; retvalue*);	N/A

C (C99)	`foo(«parameters»)`	`void foo(«parameters») { instructions }`	`type foo(«parameters») { instructions ... return value; }`	`«global declarations» int main(«int argc, char *argv[]») { instructions }`
Objective-C				
C++ (STL)				
C#				`static void Main(«string[] args») { instructions }` or `static int Main(«string[] args») { instructions }`
Java				`public static void main(String[] args) { instructions }` or `public static void main(String... args) { instructions }`
D				`int main(«char[][] args») { instructions}` or `int main(«string[] args») { instructions}` or `void main(«char[][] args») { instructions}` or `void main(«string[] args») { instructions}`
JavaScript		`function foo(«parameters») { instructions }` or `var foo = function («parameters») {instructions }` or `var foo = new Function («"parameter", ... ,"last parameter"» "instructions");`	`function foo(«parameters») { instructions ... return value; }`	N/A
Go		`func foo(«parameters») { instructions }`	`func foo(«parameters») type { instructions ... return value }`	`func main() { instructions }`

Common Lisp	**(foo** «parameters»**)**	**(defun foo** **(**«parameters»**)** instructions**)** or **(setf (symbol-function** 'symbol**)** lambda**)**	**(defun foo** **(**«parameters»**)** ... value**)**	N/A
Scheme		**(define (foo** parameters**)** instructions**)** or **(define foo (lambda** **(**parameters**)** instructions**))**	**(define (foo** parameters**)** instructions... return_value**)** or **(define foo (lambda** **(**parameters**)** instructions... return_value**))**	
ISLISP		**(defun foo** **(**«parameters»**)** instructions**)**	**(defun foo** **(**«parameters»**)** ... value**)**	

Pascal	**foo(**«**(**parameters**)**»**)**	**procedure**
		foo(«**(**parameters**)**»**);**
		«**forward;**»[a]
		«**label**
		label declarations»
		«**const**
		constant declarations»
		«**type**
		type declarations»
		«**var**
		variable declarations»
		«local function declarations»
		begin
		instructions
		end;
			function
			foo(«**(**parameters**)**»**:** type**;**
			«**forward;**»[a]
			«**label**
			label declarations»
			«**const**
			constant declarations»
			«**type**
			type declarations»
			«**var**
			variable declarations»
			«local function declarations»
			begin
			instructions**;**
			foo := value
			end;
				program name**;**
				«**label**
				label declarations»
				«**const**
				constant declarations»
				«**type**
				type declarations»
				«**var**
				variable declarations»
				«function declarations»
				begin
				instructions
				end.

Visual Basic	**Foo(**«parameters»**)**	**Sub Foo(**«parameters»**)** **instructions End Sub**	**Function Foo(**«parameters»**) As** type instructions **Foo =** value **End Function**	**Sub Main()** instructions **End Sub**
Visual Basic .NET			**Function Foo(**«parameters»**) As** type instructions **Return** value **End Function**	**Sub Main(**«**ByVal CmdArgs() As String**») **instructions End Sub** or **Function Main(**«**ByVal CmdArgs() As String**») **As Integer instructionsEnd Function**
Xojo				
Python	**foo(**«parameters»**)**	**def foo(**«parameters»**):** Tab instructions	**def foo(**«parameters»**):** Tab instructions Tab **return** value	N/A
S-Lang	**foo(**«parameters» «**;**qualifiers»**)**	**define foo (**«parameters»**) {** instructions **}**	**define foo (**«parameters»**) {** instructions ... **return** value; **}**	**public define slsh_main () {** instructions **}**
Fortran	**foo (**«arguments»**)** **CALL sub_foo (**«arguments»**)** [c]	**SUBROUTINE sub_foo (**«arguments»**)** **instructions END SUBROUTINE** [c]	type **FUNCTION foo (**«arguments»**)** **instructions ... foo = value END FUNCTION** [c]	**PROGRAM** main **instructions END PROGRAM**
Forth	«parameters» **FOO**	**:** **FOO** « *stack effect comment:* **(** before **--** **)** » **instructions ;**	**:** **FOO** « *stack effect comment:* **(** before **--** after **)** » **instructions ;**	N/A
PHP	**foo(**«parameters»**)**	**function foo(**«parameters»**) {** instructions **}**	**function foo(**«parameters»**) {** instructions ... **return** value; **}**	N/A
Perl	**foo(**«parameters»**)** or **&foo**«**(**parameters**)**»	**sub foo {** «**my (**parameters**) = @_;**»instructions **}**	**sub foo {** «**my (**parameters**) = @_;**»instructions... «**return**»value; **}**	
Perl 6	**foo(**«parameters»**)** or **&foo**«**(**parameters**)**»	«**multi** »**sub foo(**parameters**) {** instructions **}**	«**our** «type» »«**multi** »**sub foo(**parameters**) {** instructions... «**return**»value; **}**	
Ruby	**foo**«**(**parameters**)**»	**def foo**«**(**parameters**)**» instructions **end**	**def foo**«**(**parameters**)**» instructions «**return**» value **end**	
Scala		**def foo**«**(**parameters**)**»«**:** **Unit =**» **{** instructions **}**	**def foo**«**(**parameters**)**»«**:** type» **= {** instructions ... «**return**» value **}**	**def main(args: Array[String]) {** instructions **}**

<table>
<tr><td>Windows PowerShell</td><td>foo «parameters»</td><td>function foo { instructions }; or function foo { «param(parameters)» instructions }</td><td>function foo «(parameters)» { instructions ... return value }; or function foo { «param(parameters)» instructions ... return value }</td><td rowspan="3">N/A</td></tr>
<tr><td>Bash shell</td><td>foo «parameters»</td><td>function foo {
instructions
}
or
foo () {
instructions
}</td><td>function foo {
instructions
return «exit_code»
}
or
foo () {
instructions
return «exit_code»
}</td></tr>
<tr><td colspan="3">• parameters
• $n ($1, $2, $3, ...)
• <b>$@ (all parameters)
• $# (the number of parameters)
• $0 (this function name)</td></tr>
<tr><td>OCaml</td><td rowspan="4">foo parameters</td><td rowspan="2">let «rec» foo parameters = instructions</td><td rowspan="2">let «rec» foo parameters = instructions... return_value</td><td rowspan="2">[<EntryPoint>]
let main args =
instructions</td></tr>
<tr><td>F#</td></tr>
<tr><td>Standard ML</td><td>fun foo parameters = (instructions)</td><td>fun foo parameters = (instructions... return_value)</td><td></td></tr>
<tr><td>Haskell</td><td>foo parameters = do
Tab ▯ instructions</td><td>foo parameters =
return_value
or
foo parameters = do
Tab ▯ instructions
Tab ▯ return value</td><td>«main :: IO ()»
main = do
instructions</td></tr>
<tr><td>Eiffel</td><td>foo («parameters»)</td><td>foo («parameters»)
require
preconditions
do
instructions
ensure
postconditions
end</td><td>foo («parameters»): type
require
preconditions
do
instructions
Result := value
ensure
postconditions
end</td><td>[b]</td></tr>
<tr><td rowspan="2">CoffeeScript</td><td>foo()</td><td>foo = -></td><td>foo = -> value</td><td rowspan="2">N/A</td></tr>
<tr><td>foo parameters</td><td>foo = () -></td><td>foo = (parameters) -> value</td></tr>
</table>

COBOL	**CALL** "foo" «**USING** parameters» «exception-handling» «**END-CALL**»[d]	«**IDENTIFICATION DIVISION.**» **PROGRAM-ID.** foo. «other divisions...» **PROCEDURE DIVISION** «**USING** parameters». instructions.	«**IDENTIFICATION DIVISION.**» **PROGRAM-ID**/**FUNCTION-ID.** foo. «other divisions...» **DATA DIVISION.** «other sections...» **LINKAGE SECTION.** «parameter definitions...» variable-to-return definition «other sections...» **PROCEDURE DIVISION** «**USING** parameters» **RETURNING** variable-to-return. instructions.	N/A
	«**FUNCTION**» foo«(«parameters»)»	N/A		

^a Pascal requires "`forward;`" for forward declarations.

^b Eiffel allows the specification of an application's root class and feature.

^c In Fortran, function/subroutine parameters are called arguments (since `PARAMETER` is a language keyword); the `CALL` keyword is required for subroutines.

^d Instead of using "`foo`", a string variable may be used instead containing the same value.

Type conversions

Where *string* is a signed decimal number:

	string to integer	string to long integer	string to floating point	integer to string	floating point to string
Ada	**Integer'Value** (string_expression)	**Long_Integer'Value** (string_expression)	**Float'Value** (string_expression)	**Integer'Image** (integer_expression)	**Float'Image** (float_expression)
ALGOL 68 with general, and then specific formats	With prior declarations and association of: **string** buf := "12345678.9012e34 "; **file** proxy; associate(proxy, buf);				
	get(proxy, ivar);	get(proxy, livar);	get(proxy, rvar);	put(proxy, ival);	put(proxy, rval);
	getf(proxy, (g, ivar));	getf(proxy, (g, livar));	getf(proxy, (g, rvar));	putf(proxy, (g, ival));	putf(proxy, ($g(width, places, exp)$, rval));
	orv	or	or	or	or
	getf(proxy, ($dddd$, ivar));	getf(proxy, ($8d$, livar));	getf(proxy, ($8d.4dE2d$, rvar));	putf(proxy, ($4d$, ival));	putf(proxy, ($8d.4dE2d$, rval));
					etc.
C (C99)	integer = **atoi**(string);	long = **atol**(string);	float = **atof**(string);	**sprintf**(string, "%i", integer);	**sprintf**(string, "%f", float);
Objective-C	integer = [string **intValue**];	long = [string **longLongValue**];	float = [string **doubleValue**];	string = [**NSString stringWithFormat**<wbr/>:@"%i", integer];	string = [**NSString stringWithFormat**<wbr/>:@"%f", float];
C++ (STL)	«**std**::»**istringstream**(string) >> number;			«**std**::»**ostringstream** o; o << number; string = o.**str**();	
C++11	integer = «std::»**stoi**(string);	long = «std::»**stol**(string);	float = «std::»**stof**(string); double = «std::»**stod**(string);	string = «std::»**to_string**(number);	
C#	integer = **int.Parse**<wbr/>(string);	long = **long.Parse**<wbr/>(string);	float = **float.Parse**<wbr/>(string); or double = **double.Parse**<wbr/>(string);	string = number**.ToString**<wbr/>();	

D	integer = std.conv.to!int<wbr/>(string)	long = std.conv.to!long<wbr/>(string)	float = std.conv.to!float<wbr/>(string) or double = std.conv.to!double<wbr/>(string)	string = std.conv.to!string<wbr/>(number)	
Java	integer = Integer.parseInt<wbr/>(string);	long = Long.parseLong<wbr/>(string);	float = Float.parseFloat<wbr/>(string); or double = Double.parseDouble<wbr/>(string);	string = Integer.toString<wbr/>(integer); or string = String.valueOf<wbr/>(integer);	string = Float.toString<wbr/>(float); or string = Double.toString<wbr/>(double);
JavaScript[a]	integer = parseInt(string);		float = parseFloat(string); or float = new Number (string) or float = Number (string) or float = +string	string = number.toString (); or string = new String (number); or string = String (number); or string = number+"";	
Go	integer, error = strconv.Atoi(string) or integer, error = strconv.ParseInt<wbr/>(string, 10, 0)	long, error = strconv.ParseInt<wbr/>(string, 10, 64)	float, error = strconv.ParseFloat<wbr/>(string, 64)	string = strconv.Itoa(integer) or string = strconv.FormatInt<wbr/>(integer, 10) or string = fmt.Sprint(integer)	string = strconv.FormatFloat<wbr/>(float) or string = fmt.Sprint<wbr/>(float)
Common Lisp	(setf integer (parse-integer string))		(setf float (read-from-string string))	(setf string (princ-to-string number))	
Scheme	(define number (string->number string))			(define string (number->string number))	
ISLISP	(setf integer (convert string <integer>))		(setf float (convert string <float>))	(setf string (convert number <string>))	
Pascal	integer := StrToInt(string);		float := StrToFloat(string);	string := IntToStr(integer);	string := FloatToStr(float);
Visual Basic Visual Basic .NET	integer = CInt(string)	long = CLng(string)	float = CSng(string) or double = CDbl(string)	string = CStr(number)	
Xojo	integer = Val(string)	long = Val(string)	double = Val(string) or double = CDbl(string)	string = CStr(number) or string = Str(number)	
Python	integer = int(string)	long = long(string)	float = float(string)	string = str(number)	
S-Lang	integer = atoi(string);	long = atol(string);	float = atof(string);	string = string(number);	
Fortran	READ(string,format) number			WRITE(string,format) number	
PHP	integer = intval(string); or integer = (int)string;		float = floatval(string); or float = (float)string;	string = "number"; or string = strval(number); or string = (string)number;	
Perl[b]	number = 0 + string;			string = "number";	
Perl 6	number = +string;			string = ~number;	
Ruby	integer = string.to_i or integer = Integer(string)		float = string.to_f or float = Float(string)	string = number.to_s	
Scala	integer = string.toInt	long = string.toLong	float = string.toFloat or double = string.toDouble	string = number.toString	
Windows PowerShell	integer = [int]string	long = [long]string	float = [float]string	string = [string]number; or string = "number"; or string = (number).ToString()	
OCaml	let integer = int_<wbr/>of_string string		let float = float_<wbr/>of_string string	let string = string_<wbr/>of_int integer	let string = string_<wbr/>of_float float

F#	`let integer = int string`	`let integer = int64 string`	`let float = float string`	`let string = string number`	
Standard ML	`val integer = Int<wbr/>.fromString string`		`val float = Real<wbr/>.fromString string`	`val string = Int<wbr/>.toString integer`	`val string = Real<wbr/>.toString float`
Haskell (GHC)	`number = read string`			`string = show number`	
COBOL	`MOVE «FUNCTION» NUMVAL(string)`[c] `TO number`			`MOVE number TO numeric-edited`	

^a JavaScript only uses floating point numbers so there are some technicalities.

^b Perl doesn't have separate types. Strings and numbers are interchangeable.

^c `NUMVAL-C` or `NUMVAL-F` may be used instead of `NUMVAL`.

Standard stream I/O

	read from	write to	
	stdin	stdout	stderr
Ada	`Get (x)`	`Put (x)`	`Put (Standard_Error, x)`
ALGOL 68	`readf(($format$, x)); or getf(stand in, ($format$, x));`	`printf(($format$, x)); or putf(stand out, ($format$, x));`	`putf(stand error, ($format$, x));`[a]
C (C99)	`scanf(format, &x); or fscanf(stdin, format, &x);`[b]	`printf( format, x); or fprintf(stdout, format, x);`[c]	`fprintf(stderr, format, x );`[d]
Objective-C	`data = [[NSFileHandle fileHandleWithStandardInput] readDataToEndOfFile];`	`[[NSFileHandle fileHandleWithStandardOutput] writeData:data];`	`[[NSFileHandle fileHandleWithStandardError] writeData:data];`
C++	`«std::»cin >> x; or «std::»getline(«std::»cin, str);`	`«std::»cout << x;`	`«std::»cerr << x; or «std::»clog << x;`
C#	`x = Console.Read(); or x = Console.ReadLine();`	`Console.Write(«format, »x); or Console.WriteLine(«format, »x);`	`Console.Error<wbr/>.Write(«format, »x); or Console.Error<wbr/>.WriteLine(«format, »x);`
D	`x = std.stdio.readln()`	`std.stdio.write(x) or std.stdio.writeln(x) or std.stdio.writef(format, x) or std.stdio.writefln(format, x)`	`stderr.write(x) or stderr.writeln(x) or std.stdio<wbr/>.writef(stderr, format, x) or std.stdio<wbr/>.writefln(stderr, format, x)`
Java	`x = System.in.read(); or x = new Scanner(System.in)<wbr/>.nextInt(); or x = new Scanner(System.in)<wbr/>.nextLine();`	`System.out.print(x); or System.out.printf(format, x); or System.out.println(x);`	`System.err.print(x); or System.err.printf(format, x); or System.err.println(x);`
Go	`fmt.Scan(&x) or fmt.Scanf(format, &x) or x = bufio.NewReader(os.Stdin)<wbr/>.ReadString('\n')`	`fmt.Println(x) or fmt.Printf(format, x)`	`fmt.Fprintln(os.Stderr, x) or fmt.Fprintf(os.Stderr, format, x)`
JavaScript Web Browser implementation		`document.write(x)`	
JavaScript Active Server Pages		`Response.Write(x)`	

JavaScript Windows Script Host	`x = WScript.StdIn.Read(chars)` or `x = WScript.StdIn.ReadLine()`	`WScript.Echo(x)` or `WScript.StdOut.Write(x)` or `WScript.StdOut.WriteLine(x)`	`WScript.StdErr.Write(x)` or `WScript.StdErr.WriteLine(x)`
Common Lisp	`(setf x (read-line))`	`(princ x)` or `(format t format x)`	`(princ x *error-output*)` or `(format *error-output* format x)`
Scheme (R^6RS)	`(define x (read-line))`	`(display x)` or `(format #t format x)`	`(display x (current-error-port))` or `(format (current-error-port) format x)`
ISLISP	`(setf x (read-line))`	`(format (standard-output) format x)`	`(format (error-output) format x)`
Pascal	`read(x);` or `readln(x);`	`write(x);` or `writeln(x);`	N/A
Visual Basic	`Input« prompt,»x`	`Print x` or `? x`	
Visual Basic .NET	`x = Console.Read()` or `x = Console.ReadLine()`	`Console.Write(«format, »x)` or `Console.WriteLine(«format, »x)`	`Console.Error<wbr/>.Write(«format, »x)` or `Console.Error<wbr/>.WriteLine(«format, »x)`
Xojo	`x = StandardInputStream.Read()` or `x = StandardInputStreame.ReadLine()`	`StandardOutputStream.Write(x)` or `StandardOutputStream.WriteLine(x)`	`StdErr.Write(x)` or `StdErr.WriteLine(x)`
Python 2.x	`x = raw_input(«prompt»)`	`print x` or `sys.stdout.write(x)`	`print >> sys.stderr, x` or `sys.stderr.write(x)`
Python 3.x	`x = input(«prompt»)`	`print(x«, end="")`	`print(x«, end="", file=sys.stderr)`
S-Lang	`fgets (&x, stdin)`	`fputs (x, stdout)`	`fputs (x, stderr)`
Fortran	`READ(*,format) variable names` **or** `READ(INPUT_UNIT,format) variable names`[e]	`WRITE(*,format) expressions` **or** `WRITE(OUTPUT_UNIT,format) expressions`[e]	`WRITE(ERROR_UNIT,format) expressions`[e]
Forth	`buffer length ACCEPT ( # chars read )` `KEY ( char )`	`buffer length TYPE` `char EMIT`	N/A
PHP	`$x = fgets(STDIN);` or `$x = fscanf(STDIN, format);`	`print x;` or `echo x;` or `printf(format, x);`	`fprintf(STDERR, format, x);`
Perl	`$x = <>;` or `$x = <STDIN>;`	`print x;` or `printf format, x;`	`print STDERR x;` or `printf STDERR format, x;`
Perl 6	`$x = $*IN.get;`	`x.print` or `x.say`	`x.note` or `$*ERR.print(x)` or `$*ERR.say(x)`
Ruby	`x = gets`	`puts x` or `printf(format, x)`	`$stderr.puts(x)` or `$stderr.printf(format, x)`
Windows PowerShell	`$x = Read-Host«« -Prompt» text»;` or `$x = [Console]::Read();` or `$x = [Console]::ReadLine()`	`x;` or `Write-Output x;` or `echo x`	`Write-Error x`
OCaml	`let x = read_int ()` or `let str = read_line ()` or `Scanf.scanf format (fun x ... -> ...)`	`print_int x` or `print_endline str` or `Printf.printf format x ...`	`prerr_int x` or `prerr_endline str` or `Printf.eprintf format x ...`
F#	`let x = System.Console<wbr/>.ReadLine()`	`printf format x ...` or `printfn format x ...`	`eprintf format x ...` or `eprintfn format x ...`
Standard ML	`val str = TextIO.inputLIne TextIO.stdIn`	`print str`	`TextIO.output (TextIO.stdErr, str)`

Haskell (GHC)	`x <- readLn` or `str <- getLine`	`print x` or `putStrLn str`	`hPrint stderr x` or `hPutStrLn stderr str`
COBOL	`ACCEPT x`	`DISPLAY x`	

^a Algol 68 additionally as the "unformatted" transput routines: *read*, *write*, *get* and *put*.

^b `gets(x)` and `fgets(x, length, stdin)` read unformatted text from stdin. Use of gets is not recommended.

^c `puts(x)` and `fputs(x, stdout)` write unformatted text to stdout.

^d `fputs(x, stderr)` writes unformatted text to stderr

^e `INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT` are defined in the `ISO_FORTRAN_ENV` module.[14]

Reading command-line arguments

	Argument values	Argument counts	Program name / Script name
Ada	`Argument (n)`	`Argument_Count`	`Command_Name`
C (C99)	`argv[n]`	`argc`	first argument
Objective-C			
C++			
C#	`args[n]`	`args.Length`	`Assembly.GetEntryAssembly()<wbr/>.Location;`
Java		`args.length`	
D			first argument
JavaScript Windows Script Host implementation	`WScript.Arguments(n)`	`WScript.Arguments.length`	`WScript.ScriptName` or `WScript.ScriptFullName`
Go	`os.Args[n]`	`len(os.Args)`	first argument
Common Lisp	*?*	*?*	*?*
Scheme (R^6RS)	`(list-ref (command-line) n)`	`(length (command-line))`	first argument
ISLISP	N/A	N/A	N/A
Pascal	`ParamStr(n)`	`ParamCount`	first argument
Visual Basic	`Command`[a]	N/A	`App.Path`
Visual Basic .NET	`CmdArgs(n)`	`CmdArgs.Length`	`[Assembly].GetEntryAssembly().Location`
Xojo	`System.CommandLine`	(string parsing)	**Application.ExecutableFile.Name**
Python	`sys.argv[n]`	`len(sys.argv)`	first argument
S-Lang	`__argv[n]`	`__argc`	first argument
Fortran	`DO i = 1,argc` **CALL GET_COMMAND_ARGUMENT (i,argv(i)) ENDDO**	`argc = COMMAND_ARGUMENT_COUNT ()`	`CALL GET_COMMAND_ARGUMENT (0,progname)`
PHP	`$argv[n]`	`$argc`	first argument
Bash shell	`$n` (`$1, $2, $3, ...`) `$@` (all arguments)	`$#`	`$0`
Perl	`$ARGV[n]`	`scalar(@ARGV)`	`$0`

Perl 6	`@*ARGS[n]`	`@*ARGS.elems`	`$PROGRAM_NAME`
Ruby	`ARGV[n]`	`ARGV.size`	`$0`
Windows PowerShell	`$args[n]`	`$args.Length`	`$MyInvocation.MyCommand<wbr/>.Name`
OCaml	`Sys.argv.(n)`	`Array.length Sys.argv`	first argument
F#	`args.[n]`	`args.Length`	`Assembly.GetEntryAssembly()<wbr/>.Location`
Standard ML	`List.nth (CommandLine<wbr/>.arguments (), n)`	`length (CommandLine<wbr/>.arguments ())`	`CommandLine.name ()`
Haskell (GHC)	`do { args <- System.getArgs; return args !! n }`	`do { args <- System.getArgs; return length args }`	`System.getProgName`
COBOL	[b]		N/A

- **^a** The command-line arguments in Visual Basic are not separated. A split function **Split**(string) is required for separating them.
- **^b** The COBOL standard does not include any way to access command-line arguments but common compiler-extensions for accessing them include defining parameters for the main program or using `ACCEPT` statements.

Execution of commands

	Shell command	Execute program	Replace current program with new executed program
Ada	Not part of the language standard. Commonly done by compiler provided packages or by interfacing to C or Posix.[15]		
C	`system("command");`		`execl(path, args); or execv(path, arglist);`
C++			
Objective-C		`[NSTask launchedTaskWithLaunchPath:(NSString *)path arguments:(NSArray *)arguments];`	
C#		`System.Diagnostics<wbr/>.Process.Start(path, argstring);`	
F#			
Go		`exec.Run(path, argv, envv, dir, exec.DevNull, exec.DevNull, exec.DevNull)`	`os.Exec(path, argv, envv)`
Visual Basic	`Interaction.Shell(command «, WindowStyle» «, isWaitOnReturn»)`		
Visual Basic .NET	`Microsoft.VisualBasic<wbr/>.Interaction.Shell(command «, WindowStyle» «, isWaitOnReturn»)`	`System.Diagnostics<wbr/>.Process.Start(path, argstring)`	
Xojo	`Shell.Execute(command «, Parameters»)`	`FolderItem.Launch(parameters, activate)`	N/A
D	`std.process.system("command");`		`std.process.execv(path, arglist);`
Java		`Runtime.exec(command); or new ProcessBuilder(command).start();`	

JavaScript Windows Script Host implementation	`WScript.CreateObject ("WScript.Shell").Run(command «, WindowStyle» «, isWaitOnReturn»);`	`WshShell.Exec(command)`	
Common Lisp	`(shell command)`		
Scheme	`(system command)`		
ISLISP	N/A	N/A	N/A
Pascal	`system(command);`		
OCaml	`Sys.command command, Unix.open_process_full command env (stdout, stdin, stderr),...`	`Unix.create_process prog args new_stdin new_stdout new_stderr, ...`	`Unix.execv prog args` or `Unix.execve prog args env`
Standard ML	`OS.Process.system command`	`Unix.execute (path, args)`	`Posix.Process.exec (path, args)`
Haskell (GHC)	`System.system command`	`System.Process<wbr/>.runProcess pathargs ...`	`Posix.Process<wbr/>.executeFile path True args ...`
Perl	`system(command)` or `$output = `command`` or `$output = qx(command)`		`exec(path, args)`
Ruby	`system(command)` or `output = `command``		`exec(path, args)`
PHP	`system(command)` or `$output = `command`` or `exec(command)` or `passthru(command)`		
Python	`os.system(command)` or `subprocess.Popen(command)`		`os.execv(path, args)`
S-Lang	`system(command)`		
Fortran	`CALL SYSTEM (command, status)` or `status = SYSTEM (command)` [a]		
Windows PowerShell	`[Diagnostics.Process]::Start(command)`	«`Invoke-Item` »`program arg1 arg2 ...`	
Bash shell	`output=`command`` or `output=$(command)`	`program arg1 arg2 ...`	

^**a** Compiler-dependent extension.[16]

References

[1] http://en.wikipedia.org/w/index.php?title=Template:Programming_language_comparisons&action=edit

[2] Ada Reference Manual - Language and Standard Libraries; ISO/IEC 8652:201x (E), http://www.ada-auth.org/standards/12rm/RM-Final.pdf

[3] http://www.lispworks.com/documentation/HyperSpec/Front/index.htm

[4] http://www.islisp.info/specification.html

[5] http://fortranwiki.org/fortran/show/selected_int_kind

[6] 8.5 The Number Type (http://www.mozilla.org/js/language/E262-3.pdf)

[7] http://fortranwiki.org/fortran/show/selected_real_kind

[8] http://www.gnu.org/software/libc/manual/html_node/Complex-Numbers.html#Complex-Numbers

[9] http://rosettacode.org/wiki/Enumerations#ALGOL_68

[10] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in

[11] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for...of

[12] https://sites.google.com/site/truetryforvisualbasic/

[13] example (http://rosettacode.org/wiki/Prime_decomposition#ALGOL_68)

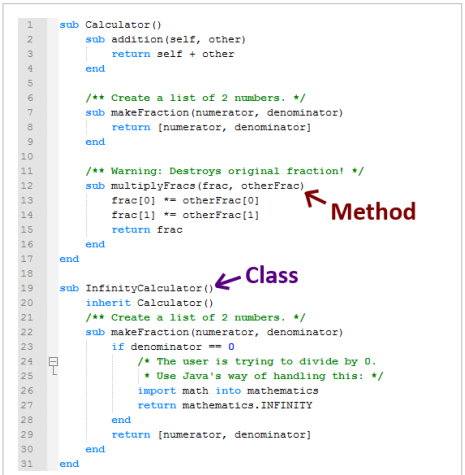
[14] http://fortranwiki.org/fortran/show/iso_fortran_env

[15] http://rosettacode.org/wiki/Execute_a_system_command#Ada
[16] http://gcc.gnu.org/onlinedocs/gfortran/SYSTEM.html#SYSTEM

Computer program

"Computer program code" and "Software code" redirect here. For their source form, see source code. For the machine-executable code, see machine code. For the TV programme, see The Computer Programme.

A **computer program**, or just a **program**, is a sequence of instructions, written to perform a specified task with a computer. A computer requires programs to function, typically executing the program's instructions in a central processor. The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, from which executable programs are derived (e.g., compiled), enables a programmer to study and develop its algorithms. A collection of computer programs and related data is referred to as the software.



A computer program written in an object-oriented style.

Computer source code is typically written by computer programmers. Source code is written in a programming language that usually follows one of two main paradigms: imperative or declarative programming. Source code may be converted into an executable file (sometimes called an executable program or a binary) by a compiler and later executed by a central processing unit. Alternatively, computer programs may be executed with the aid of an interpreter, or may be embedded directly into hardware.

Computer programs may be ranked along functional lines: system software and application software. Two or more computer programs may run simultaneously on one computer from the perspective of the user, this process being known as multitasking.

Programming

Main article: Computer programming

```c
#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Source code of a Hello World program written in the C programming language

```java
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Source code of a Hello World program written in the Java programming language

```
using System;

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
```

Source code of a Hello World program written in the C# programming language

Computer programming is the iterative process of writing or editing source code. Editing source code involves testing, analyzing, refining, and sometimes coordinating with other programmers on a jointly developed program. A person who practices this skill is referred to as a computer programmer, software developer, and sometimes coder.

The sometimes lengthy process of computer programming is usually referred to as software development. The term software engineering is becoming popular as the process is seen as an engineering discipline.

Paradigms

Computer programs can be categorized by the programming language paradigm used to produce them. Two of the main paradigms are imperative and declarative.

Programs written using an imperative language specify an algorithm using declarations, expressions, and statements. A declaration couples a variable name to a datatype. For example: `var x: integer;` . An expression yields a value. For example: `2 + 2` yields 4. Finally, a statement might assign an expression to a variable or use the value of a variable to alter the program's control flow. For example: `x := 2 + 2; if x = 4 then do_something();`. One criticism of imperative languages is the side effect of an assignment statement on a class of variables called non-local variables.

Programs written using a declarative language specify the properties that have to be met by the output. They do not specify details expressed in terms of the control flow of the executing machine but of the mathematical relations between the declared objects and their properties. Two broad categories of declarative languages are functional languages and logical languages. The principle behind functional languages (like Haskell) is to not allow side effects, which makes it easier to reason about programs like mathematical functions. The principle behind logical languages (like Prolog) is to define the problem to be solved — the goal — and leave the detailed solution to the Prolog system itself. The goal is defined by providing a list of subgoals. Then each subgoal is defined by further providing a list of its subgoals, etc. If a path of subgoals fails to find a solution, then that subgoal is backtracked and another path is systematically attempted.

The form in which a program is created may be textual or visual. In a visual language program, elements are graphically manipulated rather than textually specified.

Compiling or interpreting

A *computer program* in the form of a human-readable, computer programming language is called source code. Source code may be converted into an executable image by a compiler or executed immediately with the aid of an interpreter.

Either compiled or interpreted programs might be executed in a batch process without human interaction, but interpreted programs allow a user to type commands in an interactive session. In this case the programs are the separate commands, whose execution occurs sequentially, and thus together. When a language is used to give commands to a software application (such as a Unix shell or other command-line interface) it is called a scripting language.

Compilers are used to translate source code from a programming language into either object code or machine code. Object code needs further processing to become machine code, and machine code is the central processing unit's native code, ready for execution. Compiled computer programs are commonly referred to as executables, binary images, or simply as binaries — a reference to the binary file format used to store the executable code.

Interpreted computer programs — in a batch or interactive session — are either decoded and then immediately executed or are decoded into some efficient intermediate representation for future execution. BASIC, Perl, and Python are examples of immediately executed computer programs. Alternatively, Java computer programs are compiled ahead of time and stored as a machine independent code called bytecode. Bytecode is then executed on request by an interpreter called a virtual machine.

The main disadvantage of interpreters is that computer programs run slower than when compiled. Interpreting code is slower than running the compiled version because the interpreter must decode each statement each time it is loaded and then perform the desired action. However, software development may be faster using an interpreter because testing is immediate when the compiling step is omitted. Another disadvantage of interpreters is that at least one must be present on the computer during computer program execution. By contrast, compiled computer programs need no compiler present during execution.

No properties of a programming language require it to be exclusively compiled or exclusively interpreted. The categorization usually reflects the most popular method of language execution. For example, BASIC is thought of as an interpreted language and C a compiled language, despite the existence of BASIC compilers and C interpreters. Some systems use just-in-time compilation (JIT) whereby sections of the source are compiled 'on the fly' and stored for subsequent executions.

Self-modifying programs

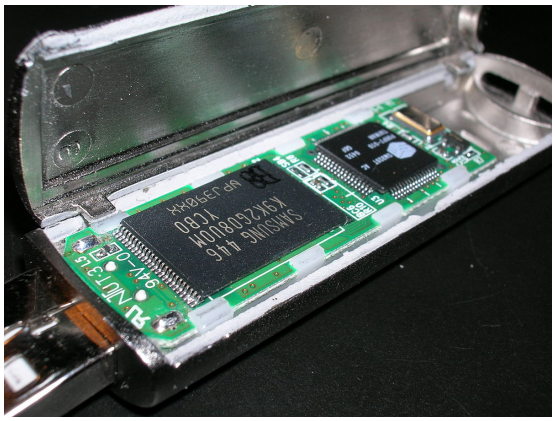
A computer program in execution is normally treated as being different from the data the program operates on. However, in some cases this distinction is blurred when a computer program modifies itself. The modified computer program is subsequently executed as part of the same program. Self-modifying code is possible for programs written in machine code, assembly language, Lisp, C, COBOL, PL/1, Prolog and JavaScript (the eval feature) among others.

Execution and storage

Typically, computer programs are stored in non-volatile memory until requested either directly or indirectly to be executed by the computer user. Upon such a request, the program is loaded into random access memory, by a computer program called an operating system, where it can be accessed directly by the central processor. The central processor then executes ("runs") the program, instruction by instruction, until termination. A program in execution is called a process. Termination is either by normal self-termination or by error — software or hardware error.

Embedded programs

Some computer programs are embedded into hardware. A stored-program computer requires an initial computer program stored in its read-only memory to boot. The boot process is to identify and initialize all aspects of the system, from processor registers to device controllers to memory contents. Following the initialization process, this initial computer program loads the operating system and sets the program counter to begin normal operations. Independent of the host computer, a hardware device might have embedded firmware to control its operation. Firmware is used when the computer program is rarely or never expected to change, or when the program must not be lost when the power is off.



The microcontroller on the right of this USB flash drive is controlled with embedded firmware.

Manual programming

Computer programs historically were manually input to the central processor via switches. An instruction was represented by a configuration of on/off settings. After setting the configuration, an execute button was pressed. This process was then repeated. Computer programs also historically were manually input via paper tape or punched cards. After the medium was loaded, the starting address was set via switches and the execute button pressed.

Switches for manual input on a Data General Nova 3

Automatic program generation

Generative programming is a style of computer programming that creates source code through generic classes, prototypes, templates, aspects, and code generators to improve programmer productivity. Source code is generated with programming tools such as a template processor or an integrated development environment. The simplest form of source code generator is a macro processor, such as the C preprocessor, which replaces patterns in source code according to relatively simple rules.

Software engines output source code or markup code that simultaneously become the input to another computer process. Application servers are software engines that deliver applications to client computers. For example, a Wiki is an application server that lets users build dynamic content assembled from articles. Wikis generate HTML, CSS, Java, and JavaScript which are then interpreted by a web browser.

Simultaneous execution

See also: Process (computing) and Multiprocessing

Many operating systems support multitasking which enables many computer programs to appear to run simultaneously on one computer. Operating systems may run multiple programs through process scheduling — a software mechanism to switch the CPU among processes often so users can interact with each program while it runs. Within hardware, modern day multiprocessor computers or computers with multicore processors may run multiple programs.

One computer program can calculate simultaneously more than one operation using threads or separate processes. Multithreading processors are optimized to execute multiple threads efficiently.

Functional categories

Computer programs may be categorized along functional lines. The main functional categories are system software and application software. System software includes the operating system which couples computer hardware with application software. The purpose of the operating system is to provide an environment in which application software executes in a convenient and efficient manner. In addition to the operating system, system software includes utility programs that help manage and tune the computer. If a computer program is not system software then it is application software. Application software includes middleware, which couples the system software with the user interface. Application software also includes utility programs that help users solve application problems, like the need for sorting.

Sometimes development environments for software development are seen as a functional category on its own, especially in the context of human-computer interaction and programming language design.Wikipedia:Please clarify Development environments gather system software (such as compilers and system's batch processing scripting languages) and application software (such as IDEs) for the specific purpose of helping programmers create new programs.

References

Further reading

- Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1, 3rd Edition*. Boston: Addison-Wesley. ISBN 0-201-89683-4.
- Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 2, 3rd Edition*. Boston: Addison-Wesley. ISBN 0-201-89684-2.
- Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 3, 3rd Edition*. Boston: Addison-Wesley. ISBN 0-201-89685-0.

External links

- Definition of "Program" (http://www.webopedia.com/TERM/P/program.html) at Webopedia
- Definition of "Computer Program" (http://dictionary.reference.com/browse/computer program) at dictionary.com

Programming language

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms.

```java
// class declaration
public class ProgrammingExample {

    // method declaration
    public void sayHello() {

        // method output
        System.out.println("Hello World!");
    }
}
```

An example of source code written in the Java programming language, which will print the message "Hello World!" to the standard output when it is compiled and then run by the Java Virtual Machine.

The earliest programming languages preceded the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos.[1] Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform), while other languages utilize other forms of program specification such as the declarative form (i.e. the desired result is specified, not how to achieve it).

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages (such as Perl) have a dominant implementation that is treated as a reference.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms.[2] Traits often considered important for what constitutes a programming language include:

Function and target

> A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation[3] or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example, PostScript programs are frequently created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language.[4] In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

Abstractions

> Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle;[5] this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.

Expressive power

> The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL-92 and Charity are examples of languages that are not Turing complete, yet often called programming languages.[6]

Markup languages like XML, HTML or troff, which define structured data, are not usually considered programming languages.[7] Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.[8]

The term *computer language* is sometimes used interchangeably with programming language.[9] However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages.[10] In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming.[11]

Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources.[12] John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.[13]

History

Main articles: History of programming languages and Programming language generations

Early developments

The first programming languages designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.[14]

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer.[15] Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. A programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

The second autocode was developed for the Mark 1 by R. A. Brooker in 1954 and was called the "Mark 1 Autocode". Brooker also developed an autocode for the Ferranti Mercury in the 1950s in conjunction with the University of Manchester. The version for the EDSAC 2 was devised by D. F. Hartley of University of Cambridge Mathematical Laboratory in 1961. Known as EDSAC 2 Autocode, it was a straight development from Mercury Autocode adapted for local circumstances, and was noted for its object code optimisation and source-language diagnostics which were advanced for the time. A contemporary but separate thread of development, Atlas Autocode was developed for the University of Manchester Atlas 1 machine.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her

team wrote a specification for an English programming language and implemented a prototype.[16] The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959.[17] Flow-Matic was a major influence in the design of COBOL, since only it and its direct descendent AIMACO were in actual use at the time.[18] The language Fortran was developed at IBM in the mid '50s, and became the first widely used high-level general purpose programming language.

Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.[19]
- PL/I, originally called NPL, was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL with block structures taken from ALGOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.[20]
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a program as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs.[21] The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

A selection of textbooks that teach programming, in languages both popular and obscure. These are only a few of the thousands of programming languages and dialects that have been designed in history.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ.

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its "here document" can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the "here document" to support multi-language programming.[22]

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

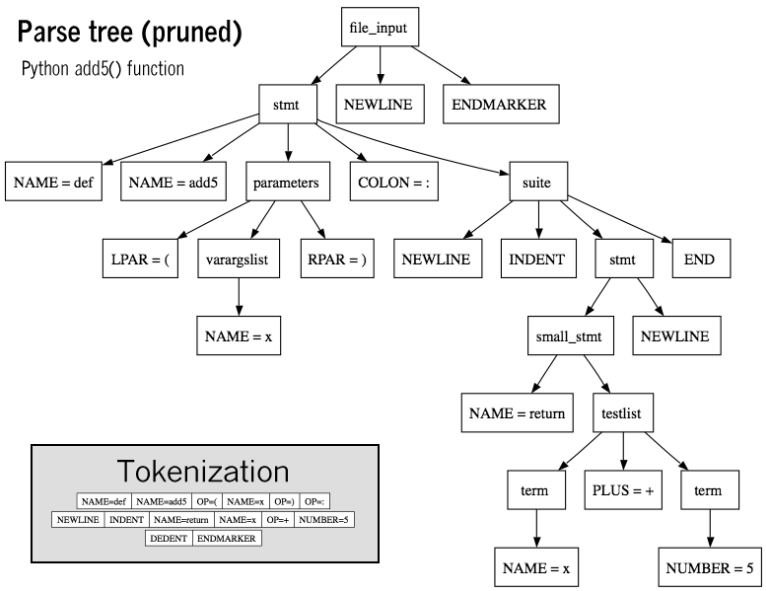
Syntax

Main article: Syntax (programming languages)

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, this article discusses textual syntax.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:



Parse tree of Python code with inset tokenization

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

```
expression ::= atom | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
```

```
list       ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: `12345`, `()` and `(a b c232 (1))`.

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs operations that are not semantically defined (the operation `*p >> 4` has no meaning for a value having a complex type and `p->im` is not defined because the value of `p` is the null pointer):

```
complex *p = NULL;
complex abs_p = sqrt(*p >> 4 + p->im);
```

If the type declaration on the first line were omitted, the program would trigger an error on compilation, as the variable "p" would not be defined. But the program would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars.[23] Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution.[24] In contrast to Lisp's macro system and Perl's `BEGIN` blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.[25]

Semantics

The term Semantics refers to the meaning of languages, as opposed to their form (syntax).

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct.[26] Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be

enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics

Main article: Semantics of programming languages

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type system

Main articles: Data type, Type system and Type safety

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type check programs, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages

A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, the data represented by "`this text between the quotes`" is a string. In most programming languages, dividing a number by a string has no meaning; most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages the meaningless operation will be detected when the program is compiled ("static" type checking), and rejected by the compiler; while in others, it will be detected when the program is run ("dynamic" type checking), resulting in a run-time exception.

A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data.

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages which are untyped include BCPL, Tcl, and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system, trading type-safety for finer control over the program's execution (see casting).

Static versus dynamic typing

In *static typing*, all expressions have their types determined prior to when the program is executed, typically at compile-time. For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.[27]

Dynamic typing, also called *latent typing*, determines the type-safety of operations at run time; in other words, types are associated with *run-time values* rather than *textual expressions*. As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Lisp, Perl, Python, JavaScript, and Ruby are dynamically typed.

Weak and strong typing

Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at run time.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression `2 * x` implicitly converts `x` to a number, and this conversion succeeds even if `x` is `null`, `undefined`, an `Array`, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

It may seem odd to some professional programmers that C could be "weakly, statically typed". However, notice that the use of the generic pointer, the **void*** pointer, does allow for casting of pointers to other pointers without needing to do an explicit cast. This is extremely similar to somehow casting an array of bytes to any kind of datatype in C without using an explicit cast, such as `(int)` or `(char)`.

Standard library and run-time system

Main article: Standard library

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so

that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the `java.lang.String` class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's `BlockContext` class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another.[28] But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many programming languages have been designed from scratch, altered to meet new needs, and combined with other languages. Many have eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role.[29] The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.[30]

Natural language programming has been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distantWikipedia:Manual of Style/Dates and numbers#Precise language and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish".[31] Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

Main article: Programming language specification

The **specification** of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a program is correct, given its source code.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX[32]). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

Main article: Programming language implementation

An **implementation** of a programming language provides a way to execute that program on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.Wikipedia:Citation needed

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness.

When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program. On the other hand, ideas about an algorithm can be communicated to humans without the precision required for execution by using pseudocode, which interleaves natural language with code written in a programming language.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the

simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language is used to give commands to a software application (such as a Unix shell or other command-line interface) it is called a scripting language.Wikipedia:Citation needed

Measuring language usage

Main article: Measuring programming language popularity

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; Fortran in scientific and engineering applications; and C in embedded applications and operating systems. Other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language – which may underestimate languages not often found in public searches[33]
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2013 the ten most popular programming languages are (in descending order by overall popularity): C, Java, PHP, JavaScript, C++, Python, Shell, Ruby, Objective-C and C#.

Taxonomies

For more details on this topic, see Categorical list of programming languages.

There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use,* with general-purpose programming languages distinguished from domain-specific programming languages. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming.[34] More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic.

An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these). Some general purpose languages were designed largely with educational goals.

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being deliberately esoteric or not.

References

[1] Ettinger, James (2004) *Jacquard's Web*, Oxford University Press

[2] In mathematical terms, this means the programming language is Turing-complete

[3] , *The scope of SIGPLAN is the theory, design, implementation, description, and application of computer programming languages - languages that permit the specification of a variety of different computations, thereby providing the user with significant control (immediate or delayed) over the computer's operation.*

[4] R. Narasimahan, Programming Languages and Computers: A Unified Metatheory, pp. 189--247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 012012108, p.193 : "a complete specification of a programming language must, by definition, include a specification of a processor--idealized, if you will--for that language." [the source cites many references to support this statement]

[5] David A. Schmidt, *The structure of typed programming languages*, MIT Press, 1994, ISBN 0-262-19349-3, p. 32

[6] , *Charity is a categorical programming language...*, *All Charity computations terminate.*

[7] XML in 10 points (http://www.w3.org/XML/1999/XML-in-10-points.html) W3C, 1999, *XML is not a programming language.*

[8] http://tobi.oetiker.ch/lshort/lshort.pdf

[9] Robert A. Edmunds, The Prentice-Hall standard glossary of computer terminology, Prentice-Hall, 1985, p. 91

[10] Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst, *Towards a General Ontology of Computer Programs* (http://www.loa-cnr.it/ICSOFT2007_final.pdf), ICSOFT 2007 (http://dblp.uni-trier.de/db/conf/icsoft/icsoft2007-1.html), pp. 163-170

[11] S.K. Bajpai, *Introduction To Computers And C Programming*, New Age International, 2007, ISBN 81-224-1379-X, p. 346

[12] R. Narasimahan, Programming Languages and Computers: A Unified Metatheory, pp. 189--247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 012012108, p.215: "[...] the model [...] for computer languages differs from that [...] for programming languages in only two respects. In a computer language, there are only finitely many names--or registers--which can assume only finitely many values--or states--and these states are not further distinguished in terms of any other attributes. [author's footnote:] This may sound like a truism but its implications are far reaching. For example, it would imply that any model for programming languages, by fixing certain of its parameters or features, should be reducible in a natural way to a model for computer languages."

[13] John C. Reynolds, *Some thoughts on teaching programming and programming languages*, SIGPLAN Notices, Volume 43, Issue 11, November 2008, p.109

[14] Rojas, Raúl, et al. (2000). "Plankalkül: The First High-Level Programming Language and its Implementation". Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text) (http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm)

[15] Sebesta, W.S Concepts of Programming languages. 2006;M6 14:18 pp.44. ISBN 0-321-33025-0

[16] Hopper (1978) p. 16.

[17] Sammet (1969) p. 316

[18] Sammet (1978) p. 204.

[19] Richard L. Wexelblat: *History of Programming Languages*, Academic Press, 1981, chapter XIV.

[20] . This comparison analyzes trends in number of projects hosted by a popular community programming repository. During most years of the comparison, C leads by a considerable margin; in 2006, Java overtakes C, but the combination of C/C++ still leads considerably.

[21] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, Akihiko Nakase (December 1994). "KLIC: A Portable Implementation of KL1" *Proc. of FGCS '94, ICOT* Tokyo, December 1994. http://www.icot.or.jp/ARCHIVE/HomePage-E.html KLIC is a portable implementation of a concurrent logic programming language KL1.

[22] Wall, *Programming Perl* ISBN 0-596-00027-8 p. 66

[23] Section 2.2: Pushdown Automata, pp.101–114.

[24] Jeffrey Kegler, " Perl and Undecidability (http://www.jeffreykegler.com/Home/perl-and-undecidability)", *The Perl Review*. Papers 2 and 3 prove, using respectively Rice's theorem and direct reduction to the halting problem, that the parsing of Perl programs is in general undecidable.

[25] Marty Hall, 1995, Lecture Notes: Macros (http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.html), PostScript version (http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.ps)

[26] Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 18–19

[27] Specifically, instantiations of generic types are inferred for certain expression forms. Type inference in Generic Java—the research language that provided the basis for Java 1.5's bounded parametric polymorphism extensions—is discussed in two informal manuscripts from the Types mailing list: Generic Java type inference is unsound (http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00849.html) (Alan Jeffrey, 17 December 2001) and Sound Generic Java type inference (http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00921.html) (Martin Odersky, 15 January 2002). C#'s type system is similar to Java's, and uses a similar partial type inference scheme.

[28] Steven R. Fischer, *A history of language*, Reaktion Books, 2003, ISBN 1-86189-080-X, p. 205

[29] IBM in first publishing PL/I, for example, rather ambitiously titled its manual *The universal programming language PL/I* (IBM Library; 1966). The title reflected IBM's goals for unlimited subsetting capability: *PL/I is designed in such a way that one can isolate subsets from it satisfying the requirements of particular applications.* (). Ada and UNCOL had similar early goals.

[30] Frederick P. Brooks, Jr.: *The Mythical Man-Month,* Addison-Wesley, 1982, pp. 93-94

[31] Dijkstra, Edsger W. On the foolishness of "natural language programming." (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html) EWD667.

[32] ANSI — Programming Language Rexx, X3-274.1996

[33] Bieman, J.M.; Murdock, V., Finding code on the World Wide Web: a preliminary investigation, Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, 2001

[34] Carl A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992, ISBN 0-262-57095-5, p. 1

Further reading

See also: History of programming languages: Further reading|History of programming languages § Further reading|History of programming languages: Further reading

- Abelson, Harold; Sussman, Gerald Jay (1996). *Structure and Interpretation of Computer Programs* (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html) (2nd ed.). MIT Press.
- Raphael Finkel: *Advanced Programming Language Design* (http://www.nondot.org/sabre/Mirrored/AdvProgLangDesign/), Addison Wesley 1995.
- Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes: *Essentials of Programming Languages*, The MIT Press 2001.
- Maurizio Gabbrielli and Simone Martini: "Programming Languages: Principles and Paradigms", Springer, 2010.
- David Gelernter, Suresh Jagannathan: *Programming Linguistics*, The MIT Press 1990.
- Ellis Horowitz (ed.): *Programming Languages, a Grand Tour* (3rd ed.), 1987.
- Ellis Horowitz: *Fundamentals of Programming Languages*, 1989.
- Shriram Krishnamurthi: *Programming Languages: Application and Interpretation*, online publication (http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/).
- Bruce J. MacLennan: *Principles of Programming Languages: Design, Evaluation, and Implementation*, Oxford University Press 1999.
- John C. Mitchell: *Concepts in Programming Languages*, Cambridge University Press 2002.
- Benjamin C. Pierce: *Types and Programming Languages*, The MIT Press 2002.
- Terrence W. Pratt and Marvin V. Zelkowitz: *Programming Languages: Design and Implementation* (4th ed.), Prentice Hall 2000.
- Peter H. Salus. *Handbook of Programming Languages* (4 vols.). Macmillan 1998.
- Ravi Sethi: *Programming Languages: Concepts and Constructs*, 2nd ed., Addison-Wesley 1996.
- Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers 2005.
- Robert W. Sebesta: *Concepts of Programming Languages*, 9th ed., Addison Wesley 2009.
- Franklyn Turbak and David Gifford with Mark Sheldon: *Design Concepts in Programming Languages*, The MIT Press 2009.
- Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*, The MIT Press 2004.
- David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall 1990.
- David A. Watt and Muffy Thomas. *Programming Language Syntax and Semantics*. Prentice Hall 1991.
- David A. Watt. *Programming Language Processors*. Prentice Hall 1993.

- David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons 2004.

External links

- 99 Bottles of Beer (http://www.99-bottles-of-beer.net/) A collection of implementations in many languages.
- Computer Programming Languages (http://www.dmoz.org/Computers/Programming/Languages) at DMOZ

Abstraction

In computer science, **abstraction** is the process of separating ideas from specific instances of those ideas at work. Computational structures are defined by their meanings (semantics), while hiding away the details of how they work. Abstraction tries to factor out details from a common pattern so that programmers can work close to the level of human thought, leaving out details which matter in practice, but are immaterial to the problem being solved. For example, a system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer; low-level abstraction layers expose details of the computer hardware where the program runs, while high-level layers deal with the business logic of the program.

Abstraction captures only those details about an object that are relevant to the current perspective; in both computing and in mathematics, numbers are concepts in programming languages. Numbers can be represented in myriad ways in hardware and software, but, irrespective of how this is done, numerical operations will obey identical rules.

Abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subprograms and related concepts control flows
- Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype.

One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.

The same abstract definition can be used as a common interface for a family of objects with different implementations and behaviors but which share the same meaning. The inheritance mechanism in object-oriented programming can be used to define an abstract class as the common interface.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called the abstraction principle.

Rationale

Computing mostly operates independently of the concrete world: The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentration on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's Tenth Rule is an aphorism on how such an architecture is both inevitable and complex.

A central form of abstraction in computing is language abstraction: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer languages* can be processed with a computer. An example of this abstraction process is the generational development of programming languages from the machine language to the assembly language and the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues for example in scripting languages and domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include the subroutine, the module, and the software component. Some other abstractions such as software design patterns and architectural styles remain invisible to a programming language and operate only in the design of a system.

Some abstractions try to limit the breadth of concepts a programmer needs by completely hiding the abstractions that in turn are built on. The software engineer and writer Joel Spolsky has criticised these efforts by claiming that all abstractions are *leaky* — that they can never completely hide the details below Wikipedia:Citation needed; however this does not negate the usefulness of abstraction. Some abstractions are designed to interoperate with others, for example a programming language may contain a foreign function interface for making calls to the lower-level language. Data abstraction is the separation between the specification of data object and its implementation.

Language features

Programming languages

Main article: Programming language

Different programming languages provide different types of abstraction, depending on the intended applications for the language. For example:

- In object-oriented programming languages such as C++, Object Pascal, or Java, the concept of **abstraction** has itself become a declarative statement - using the keywords `virtual` (in C++) or `abstract` and `interface` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.
- Functional programming languages commonly exhibit abstractions related to functions, such as lambda abstractions (making a term into a function of some variable), higher-order functions (parameters are functions), bracket abstraction (making a term into a function of a variable).
- Modern Lisps such as Clojure, Scheme and Common Lisp support macro systems to allow syntactic abstraction. This allows a Lisp programmer to eliminate boilerplate code, abstract away tedious function call sequences, implement new control flow structures, implement or even build Domain Specific Languages (DSLs), which allow domain-specific concepts to be expressed in some optimised way. All of these, when used correctly, improve both the programmer's efficiency and the clarity of the code by making the intended purpose more explicit. A consequence of syntactic abstraction is also that any Lisp dialect and in fact almost any programming language can, in principle, be implemented in any modern Lisp with significantly reduced (but still non-trivial in some cases) effort when compared to "more traditional" programming languages such as Python, C or Java.

Specification methods

Main article: Formal specification

Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE, CASL);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages

Main article: Specification language

Specification languages generally rely on abstractions of one kind or another, since specifications are typically defined earlier in a project, (and at a more abstract level) than an eventual implementation. The UML specification language, for example, allows the definition of *abstract* classes, which remain abstract during the architecture and specification phase of the project.

Control abstraction

Main article: Control flow

Programming languages offer control abstraction as one of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from one location of the memory to another location and producing the sum of two sequences of bits. Programming languages allow this to be done in the higher level. For example, consider this statement written in a Pascal-like fashion:

```
a := (1 + 2) * 5
```

To a human, this seems a fairly simple and obvious calculation (*"one plus two is three, times five is fifteen"*). However, the low-level steps necessary to carry out this evaluation, and return the value "15", and then assign that value to the variable "a", are actually quite subtle and complex. The values need to be converted to binary representation (often a much more complicated task than one would think) and the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of one register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of "15" to the variable labeled "a", so that "a" can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label and the resultant location in physical or virtual memory, storing the binary representation of "15" to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time she simply wanted to add or multiply a couple of numbers and assign the result to a variable. Such duplication of effort has two serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed
2. it forces the programmer to program for the particular hardware and instruction set

Structured programming

Main article: Structured programming

Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control and interfaces between components, with reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points and (where possible) to have single exit points from functions and procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships and at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on and off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen and the database interface (which

may itself be a standalone third party library or a statically linked set of library routines).
- Either the database or the payroll application also has to initiate the process of exchanging data with between ship and shore, and that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of one component and its assorted internal methods from the others. Object-oriented programming embraced and extended this concept.

Data abstraction

Main article: Abstract data type

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. As one way to look at this: the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include Ada and Modula-2. Object-oriented languages are commonly claimedWikipedia:Manual of Style/Words to watch#Unsupported attributions to offer data abstraction; however, their inheritance concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the Fragile binary interface problem.

Abstraction in object oriented programming

Main article: Object (computer science)

In object-oriented programming theory, **abstraction** involves the facility to define objects that represent abstract "actors" that can perform work, report on and change their state, and "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, and standardizing the way that different data types interact, is the beginning of **abstraction**. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various object-oriented programming languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of one type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Common Lisp Object System or Self, for example, feature less of a class-instance distinction and more use of delegation for polymorphism. Individual objects and functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates and overloading and other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter the need to support abstract nouns in code - all programming relies on an ability to abstract verbs as functions, nouns as data structures, and either as processes.

Consider for example a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger and feeding. It defines an `Animal` class to represent both the state of the animal and its functions:

```
public class Animal extends LivingThing
{
     private Location loc;
     private double energyReserves;

     public boolean isHungry() {
         return energyReserves < 2.5;
     }
     public void eat(Food food) {
         // Consume food
         energyReserves += food.getCalories();
     }
     public void moveTo(Location location) {
         // Move to new location
         this.loc = location;
     }
}
```

With the above definition, one could create objects of type `Animal` and call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class *`Animal`* is an abstraction used in place of an actual animal, *`LivingThing`* is a further abstraction (in this case a generalisation) of *`Animal`*.

If one requires a more differentiated hierarchy of animals — to differentiate, say, those who provide milk from those who provide nothing except meat at the end of their lives — that is an intermediary level of abstraction, probably DairyAnimal (cows, goats) who would eat foods suitable to giving good milk, and MeatAnimal (pigs, steers) who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so s/he could concentrate instead on the feeding schedule. The two classes could be related using inheritance or stand alone, and the programmer could define varying degrees of polymorphism between the two types. These facilities tend to vary

drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design

Main article: Object-oriented design

Decisions regarding what to abstract and what to keep under the control of the coder become the major concern of object-oriented design and domain analysis—actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time and budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs and cows and their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available and thus there is no reason to code the type of food into the class itself, and the design is a single simple Animal class of which pigs and cows are instances with the same functions. A decision to differentiate DairyAnimal would change the detailed analysis but the domain and legacy analysis would be unchanged—thus it is entirely under the control of the programmer, and we refer to abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, **abstraction** refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. For instance, one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a **concrete** (more precise) model of execution.

Abstraction may be **exact** or **faithful** with respect to a property if one can answer a question about the property equally well on the concrete or abstract model. For instance, if we wish to know what the result of the evaluation of a mathematical expression involving only integers +, -, ×, is worth modulo n, we need only perform all operations modulo n (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily **exact**, should be **sound**. That is, it should be possible to get sound answers from them—even though the abstraction may simply yield a result of undecidability. For instance, we may abstract the students in a class by their minimal and maximal ages; if one asks whether a certain person belongs to that class, one may simply compare that person's age with the minimal and maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction and other characteristics of the design, and how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because non-trivial properties of computer programs are essentially undecidable (see Rice's theorem). As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

Levels of abstraction

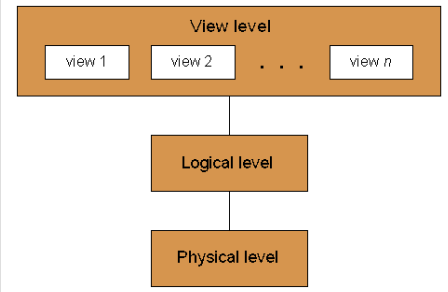
Main article: Abstraction layer

Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information and processes, but uses a system of expression involving a unique set of objects and compositions that apply only to a particular domain. [1] Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. For example, gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications and operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems

Main article: Database management system

Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels:



Data abstraction levels of a database system

Physical level: The highest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.

Logical level: The next higher level of abstraction describes *what* data the database stores, and what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. This referred to as Physical Data Independence. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.

View level: The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Layered architecture

Main article: Abstraction layer

The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction
- support the portability of software artefacts (model-based ideally)

Systems design and business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). It is a technique used in designing computer software, hardware, and communications in which system or network components are isolated in layers so that changes can be made in one layer without affecting the others.

References

[1] Luciano Floridi, *Levellism and the Method of Abstraction* (http://www.cs.ox.ac.uk/activities/ieg/research_reports/ieg_rr221104.pdf) IEG – Research Report 22.11.04

Further reading

- Harold Abelson; Gerald Jay Sussman; Julie Sussman (25 July 1996). *Structure and Interpretation of Computer Programs* (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html) (2 ed.). MIT Press. ISBN 978-0-262-01153-2. Retrieved 22 June 2012.
- Lorenza Saitta; Jean-Daniel Zucker (2013). *Abstraction in Artificial Intelligence and Complex Systems* (http://link.springer.com/book/10.1007/978-1-4614-7052-6) (1 ed.). Springer-Verlag. ISBN 978-1-4614-7051-9.
- Spolsky, Joel (11 November 2002). "The Law of Leaky Abstractions" (http://www.joelonsoftware.com/articles/LeakyAbstractions.html). *Joel on Software*.
- Abstraction/information hiding (http://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L08-abstraction/08_abstraction.html) - CS211 course, Cornell University.
- Gorodinski, Lev (31 May 2012). "Abstractions" (http://gorodinski.com/blog/2012/05/31/abstractions/).

External links

- SimArch (https://sites.google.com/site/simulationarchitecture/) example of layered architecture for distributed simulation systems.

Programmer

This article is about people who write computer software. For other uses, see Programmer (disambiguation).

For someone who performs coding in the social sciences, see Coding (social sciences).

"Coder" redirects here. For other uses, see Encoder.

Student programmers at the Technische Hochschule in Aachen, Germany in 1970

A **programmer**, **computer programmer**, **developer**, **coder**, or **software engineer** is a person who writes computer software. The term *computer programmer* can refer to a specialist in one area of computer programming or to a generalist who writes code for many kinds of software. One who practices or professes a formal approach to programming may also be known as a programmer analyst. A programmer's primary computer language (C, C++, C#, Java, Lisp, Python, etc.) is often prefixed to the above titles, and those who work in a web environment often prefix their titles with *Web*. The term *programmer* can be used to refer to a software developer, Web developer, mobile applications developer, embedded firmware developer, software engineer, computer scientist, or software analyst. However, members of these professions typicallyWikipedia:Citation needed possess other software engineering skills, beyond programming; for this reason, the term *programmer*, or *code monkey*, is sometimes considered an insulting or derogatory oversimplification of these other professions. This has sparked much debate amongst developers, analysts, computer scientists, programmers, and outsiders who continue to be puzzled at the subtle differences in the definitions of these occupations.

Ada Lovelace, the first computer programmer.

British countess and mathematician Ada Lovelace is considered the first computer programmer, as she was the first to write and publish an algorithm intended for implementation on Charles Babbage's analytical engine, in October 1842, intended for the calculation of Bernoulli numbers.[1] Lovelace was also the first person to comment on the potential for computers to be used for purposes other than computing calculations. Because Babbage's machine was never completed to a functioning standard in her time, she never saw her algorithm run.

The first person to run a program on a functioning modern electronically based computer was computer scientist Konrad Zuse, in 1941.

The ENIAC programming team, consisting of Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas and Ruth Lichterman were the first regularly working programmers.

International Programmers' Day is celebrated annually on 7 January. In 2009, the government of Russia decreed a professional annual holiday known as Programmers' Day to be celebrated on 13 September (12 September in leap years). It had also been an unofficial international holiday before that.

Nature of the work

Some of this section is from the Occupational Outlook Handbook [2], 2006–07 Edition, which is in the public domain as a work of the United States Government.

Computer programmers write, test, debug, and maintain the detailed instructions, called computer programs, that computers must follow to perform their functions. Programmers also conceive, design, and test logical structures for solving problems by computer. Many technical innovations in programming — advanced computing technologies and sophisticated new languages and programming tools — have redefined the role of a programmer and elevated much of the programming work done today. Job titles and descriptions may vary, depending on the organization.

Programmers in the Yandex headquarters.

Programmers work in many settings, including corporate information technology ("IT") departments, big software companies, and small service firms. Many professional programmers also work for consulting companies at client sites as contractors. Licensing is not typically required to work as a programmer, although professional certifications are commonly held by programmers. Programming is widely considered a profession (although someWikipedia:Avoid weasel words authorities disagree on the grounds that only careers with legal licensing requirements count as a profession).

Programmers' work varies widely depending on the type of business for which they are writing programs. For example, the instructions involved in updating financial records are very different from those required to duplicate conditions on an aircraft for pilots training in a flight simulator. Although simple programs can be written in a few hours, programs that use complex mathematical formulas whose solutions can only be approximated or that draw data from many existing systems may require more than a year of work. In most cases, several programmers work together as a team under a senior programmer's supervision.

Programmers write programs according to the specifications determined primarily by more senior programmers and by systems analysts. After the design process is complete, it is the job of the programmer to convert that design into a logical series of instructions that the computer can follow. The programmer codes these instructions in one of many programming languages. Different programming languages are used depending on the purpose of the program. COBOL, for example, is commonly used for business applications that typically run on mainframe and midrange computers, whereas Fortran is used in science and engineering. C++ is widely used for both scientific and business applications. Java, C#, VB and PHP are popular programming languages for Web and business applications. Programmers generally know more than one programming language and, because many languages are similar, they often can learn new languages relatively easily. In practice, programmers often are referred to by the language they know, e.g. as *Java programmers*, or by the type of function they perform or environment in which they work: for example, *database programmers*, *mainframe programmers*, or Web developers.

When making changes to the source code that programs are made up of, programmers need to make other programmers aware of the task that the routine is to perform. They do this by inserting comments in the source code so that others can understand the program more easily. To save work, programmers often use libraries of basic code that can be modified or customized for a specific application. This approach yields more reliable and consistent programs and increases programmers' productivity by eliminating some routine steps.

Testing and debugging

Programmers test a program by running it and looking for bugs (errors). As they are identified, the programmer usually makes the appropriate corrections, then rechecks the program until an acceptably low level and severity of bugs remain. This process is called testing and debugging. These are important parts of every programmer's job. Programmers may continue to fix these problems throughout the life of a program. Updating, repairing, modifying, and expanding existing programs is sometimes called *maintenance programming*. Programmers may contribute to user guides and online help, or they may work with technical writers to do such work.

Application versus system programming

Computer programmers often are grouped into two broad types: application programmers and systems programmers. Application programmers write programs to handle a specific job, such as a program to track inventory within an organization. They also may revise existing packaged software or customize generic applications which are frequently purchased from independent software vendors. Systems programmers, in contrast, write programs to maintain and control computer systems software, such as operating systems and database management systems. These workers make changes in the instructions that determine how the network, workstations, and CPU of the system handle the various jobs they have been given and how they communicate with peripheral equipment such as printers and disk drives.

Types of software

Programmers in software development companies may work directly with experts from various fields to create software – either programs designed for specific clients or packaged software for general use – ranging from video games to educational software to programs for desktop publishing and financial planning. Programming of packaged software constitutes one of the most rapidly growing segments of the computer services industry. Some companies or organizations – even small ones – have set up their own IT team to ensure the design and development of in-house software to answer to very specific needs from their internal end-users, especially when existing software are not suitable or too expensive. This is for example the case in research laboratories.Wikipedia:Citation needed

In some organizations, particularly small ones, workers commonly known as *programmer analysts* are responsible for both the systems analysis and the actual programming work. The transition from a mainframe environment to one that is based primarily on personal computers (PCs) has blurred the once rigid distinction between the programmer and the user. Increasingly, adept end users are taking over many of the tasks previously performed by programmers. For example, the growing use of packaged software, such as spreadsheet and database management software packages, allows users to write simple programs to access data and perform calculations.Wikipedia:Citation needed

In addition, the rise of the Internet has made web development a huge part of the programming field. Currently more software applications are web applications that can be used by anyone with a web browser. Examples of such applications include the Google search service, the Hotmail e-mail service, and the Flickr photo-sharing service.Wikipedia:Citation needed

Programming editors, also known as source code editors, are text editors that are specifically designed for programmers or developers for writing the source code of an application or a program. Most of these editors include features useful for programmers, which may include color syntax highlighting, auto indentation, auto-complete, bracket matching, syntax check, and allows plug-ins. These features aid the users during coding, debugging and testing.Wikipedia:Citation needed

Globalization

Market changes in the UK

According to BBC, 17% of computer science students could not find work in their field 7 months after graduation in 2009 which was the highest rate of the university subjects surveyed while 0% of medical students were unemployed in the same survey.[3] The UK category system does, however, class such degrees as information technology and game design as 'computer science', industries in which jobs can be extremely difficult to find, somewhat inflating the actual figure.[4]

Market changes in the US

Computer programming, offshore outsourcing, and Foreign Worker Visas became a controversial topic after the crash of the dot com bubble left many programmers without work or with lower wages. Programming was even mentioned in the 2004 US Presidential debate on the topic of offshore outsourcing.Wikipedia:Citation needed

Large companies claim there is a skills shortage with regard to programming talent. However, US programmers and unions counter that large companies are exaggerating their case in order to obtain cheaper programmers from developing countries and to avoid paying for training in very specific technologies.[5]

Enrolment in computer-related degrees in US has dropped recently due to lack of general interests in science and mathematics and also out of an apparent fear that programming will be subject to the same pressures as manufacturing and agriculture careers.Wikipedia:Citation needed This situation has resulted in confusion about whether the US economy is entering a "post-information age" and the nature of US comparative advantages. Technology and software jobs were supposed to be the replacement for factory and agriculture jobs lost to cheaper foreign labor, but if those are subject to free trade losses, then the nature of the next generation of replacement careers is not clear at this point.Wikipedia:Citation needed

References

[1] J. Fuegi and J. Francis, "Lovelace & Babbage and the creation of the 1843 'notes'." Annals of the History of Computing 25 #4 (October–December 2003): 19, 25. Digital Object Identifier (http://dx.doi.org/10.1109/MAHC.2003.1253887)

[2] http://www.bls.gov/oco/ocos110.htm

[3] "'One in 10' UK graduates unemployed" (http://www.bbc.co.uk/news/10477551) from the BBC

[4] (http://www.plymouth.ac.uk/pages/view.asp?page=23727) ATAS classifications (University of Plymouth)

[5] (http://heather.cs.ucdavis.edu/MigLtrs.pdf) Migration Letters, Volume: 10, No: 2, pp. 211 – 228 ISSN: 1741-8984 & eISSN: 1741-8992

Further reading

- Weinberg, Gerald M., *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971
- An experiential study of the nature of programming work: Lucas, Rob. "Dreaming in Code" (http://www.newleftreview.org/?view=2836) *New Left Review* 62, March–April 2010, pp. 125–132.

External links

- "The Future of IT Jobs in America" article (http://www.ideosphere.com/fx-bin/Claim?claim=ITJOBS)
- How to be a programmer (http://samizdat.mines.edu/howto/HowToBeAProgrammer.html) - An overview of the challenges of being a programmer
- The US Department of Labor's description of " Computer Programmers (http://www.bls.gov/ooh/computer-and-information-technology/computer-programmers.htm)"

Language primitive

In computing, **language primitives** are the simplest elements available in a programming language. A primitive is the smallest 'unit of processing' available to a programmer of a particular machine, or can be an atomic element of an expression in a language.

Primitives are units with a meaning, i.e. a semantic value in the language. Thus they're different from tokens in a parser, which are the minimal elements of syntax.

Machine level primitives

A machine instruction, usually generated by an assembler program, is often considered the smallest unit of processing although this is not always the case. It typically performs what is perceived to be one single operation such as copying a byte or string of bytes from one memory location to another or adding one processor register to another.

Micro code primitives

Many of today's computers, however, actually embody an even lower unit of processing known as microcode which interprets the "machine code" and it is then that the microcode instructions would be the *genuine* primitives. These instructions would typically be available for modification only by the hardware vendors programmers.

High level language primitives

A high-level programming language (*HLL*) program is composed of discrete statements and primitive data types that may also be *perceived* to perform a single operation or represent a single data item, but at a more abstract level than those provided by the machine. Copying a data item from one location to another may actually involve many machine instructions that, for instance,

- calculate the address of both operands in memory, based on their positions within a data structure,
- convert from one data type to another

before finally

- performing the final store operation to the target destination.

Some HLL statements, particularly those involving loops, can generate thousands or even millions of primitives in a low level language - which comprise the genuine instruction path length the processor has to execute at the lowest level. This perception has been referred to as the "Abstraction penalty"

Interpreted language primitives

An interpreted language statement has similarities to the HLL primitives but with a further added 'layer'. Before the statement can be executed in a manner very similar to a HLL statement, first, it has to be processed by an interpreter, a process that may involve many primitives in the target machine language.

Fourth and Fifth-generation programming language primitives

4gls and 5gls do not have a simple one-to-many correspondence from high-to-low level primitives. There are some elements of interpreted language primitives embodied in 4gl and 5gl specifications but the approach to the original problem is less a procedural language construct and are more oriented toward problem solving and systems engineering.

References

Assembly language

See the terminology section below for information regarding inconsistent use of the terms assembly and assembler.

An **assembly language** is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Each assembly language is specific to a particular computer architecture, in contrast to most high-level programming languages, which are generally portable across multiple architectures, but require interpreting or compiling.

Assembly language is converted into executable machine code by a utility program referred to as an *assembler*; the conversion process is referred to as *assembly*, or *assembling* the code.

Assembly language uses a mnemonic to represent each low-level machine instruction or operation. Typical operations require one or more operands in order to form a complete instruction, and most assemblers can therefore take labels, symbols and expressions as operands to represent addresses and other constants, freeing the programmer from tedious manual calculations. **Macro assemblers** include a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2


C000                    ORG    ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS    #STACK

                ****************************************
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

0013            RESETA  EQU    %00010011
0011            CTLREG  EQU    %00010001

C003 86 13      INITA   LDA A  #RESETA   RESET ACIA
C005 B7 80 04           STA A  ACIA
C008 86 11              LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04           STA A  ACIA

C00D 7E C0 F1           JMP    SIGNON    GO TO START OF MONITOR

                ****************************************
                * FUNCTION: INCH - Input character
                * INPUT: none
                * OUTPUT: char in acc A
                * DESTROYS: acc A
                * CALLS: none
                * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04  INCH     LDA A  ACIA      GET STATUS
C013 47                 ASR A            SHIFT RDRF FLAG INTO CARRY
C014 24 FA              BCC    INCH      RECIEVE NOT READY
C016 B6 80 05           LDA A  ACIA+1    GET CHAR
C019 84 7F              AND A  #$7F      MASK PARITY
C01B 7E C0 79           JMP    OUTCH     ECHO & RTS

                ****************************************
                * FUNCTION: INHEX - INPUT HEX DIGIT
                * INPUT: none
                * OUTPUT: Digit in acc A
                * CALLS: INCH
                * DESTROYS: acc A
                * Returns to monitor if not HEX input

C01E 8D F0     INHEX    BSR    INCH      GET A CHAR
C020 81 30              CMP A  #'0       ZERO
C022 2B 11              BMI    HEXERR    NOT HEX
C024 81 39              CMP A  #'9       NINE
C026 2F 0A              BLE    HEXRTS    GOOD HEX
C028 81 41              CMP A  #'A
C02A 2B 09              BMI    HEXERR    NOT HEX
C02C 81 46              CMP A  #'F
C02E 2E 05              BGT    HEXERR
C030 80 07              SUB A  #7        FIX A-F
C032 84 0F     HEXRTS   AND A  #$0F      CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF  HEXERR   JMP    CTRL      RETURN TO CONTROL LOOP
```

Motorola MC6800 Assembly listing, showing original assembly language and the assembled form

Key concepts

Assembler

An **assembler** is a program which creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("opcode") as well as other control bits.[1] The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.[2] The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearagement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.Wikipedia:Citation needed

Like early programming languages such as Fortran, Algol, Cobol and Lisp, assemblers have been available since the 1950s and the first generations of text based computer interfaces. However, assemblers came first as they are far simpler to write than compilers for high-level languages. This is because each mnemonic along with the addressing modes and operands of an instruction translates rather directly into the numeric representations of that particular instruction, without much context or analysis. There have also been several classes of translators and semi automatic code generators with properties similar to both assembly and high level languages, with speedcode as perhaps one of the better known examples.

Number of passes

There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
- Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was speed of assembly— often a second pass would require rewinding and rereading a tape or rereading a deck of cards. With modern computers this has ceased to be an issue. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.

High-level assemblers

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing (although available on ordinary assemblers since the late 1950s for IBM 700 series and since the 1960s for IBM/360, amongst other machines)
- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance[3]

See Language design below for more details.

Assembly language

A program written in assembly language consists of a series of (mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters. These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.

```
10110000 01100001
```

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows.

```
B0 61
```

Here, `B0` means 'Move a copy of the following value into *AL*', and `61` is a hexadecimal representation of the value 01100001, which is 97 in decimal. Intel assembly language provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

```
MOV AL, 61h        ; Load AL with 97 decimal (61 hex)
```

In some assembly languages the same mnemonic such as MOV may be used for a family of related instructions for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. Other assemblers may use separate opcodes such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

The Intel opcode 10110000 (`B0`) copies an 8-bit value into the *AL* register, while 10110001 (`B1`) moves it into *CL* and 10110010 (`B2`) does so into *DL*. Assembly language examples for these follow.

```
MOV AL, 1h         ; Load AL with immediate value 1
MOV CL, 2h         ; Load CL with immediate value 2
MOV DL, 3h         ; Load DL with immediate value 3
```

The syntax of MOV can also be more complex as the following examples show.

```
MOV EAX, [EBX]        ; Move the 4 bytes in memory at the address contained in EBX into EAX
MOV [ESI+EAX], CL ; Move the contents of CL into the byte at address ESI+EAX
```

In each case, the MOV mnemonic is translated directly into an opcode in the ranges 88-8E, A0-A3, B0-B8, C6 or C7 by an assembler, and the programmer does not have to know or remember which.

Transforming assembly language into machine code is the job of an assembler, and the reverse can at least partially be achieved by a disassembler. Unlike high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* (essentially macros) which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture has its own machine language. Computers differ in the number and type of operations they support, in the different sizes and numbers of registers, and in the representations of data in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

Multiple sets of mnemonics or assembly-language syntax may exist for a single instruction set, typically instantiated in different assembler programs. In these cases, the most popular one is usually that supplied by the manufacturer and used in its documentation.

Language design

Basic elements

There is a large degree of diversity in the way the authors of assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation (pseudo-op). A typical assembly language consists of 3 types of instruction statements that are used to define program operations:

- Opcode mnemonics
- Data definitions
- Assembly directives

Opcode mnemonics and extended mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level language. Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an opcode), and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or *opcode* plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be immediate (value coded in the instruction itself), registers specified in the instruction or implied, or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works. *Extended mnemonics* are often used to specify a combination of an opcode with a specific operand, e.g., the System/360 assemblers use **B** as an extended mnemonic for **BC** with a mask of 15 and **NOP** ("NO OPeration" - do nothing for one step) for **BC** with a mask of 0.

Extended mnemonics are often used to support specialized uses of instructions, often for purposes not obvious from the instruction name. For example, many CPU's do not have an explicit NOP instruction, but do have instructions that can be used for the purpose. In 8086 CPUs the instruction *xchg ax,ax* is used for *nop*, with *nop* being a pseudo-opcode to encode the instruction *xchg ax,ax*. Some disassemblers recognize this and will decode the *xchg ax,ax* instruction as *nop*. Similarly, IBM assemblers for System/360 and System/370 use the extended mnemonics *NOP* and *NOPR* for *BC* and *BCR* with zero masks. For the SPARC architecture, these are known as *synthetic instructions*.

Some assemblers also support simple built-in macro-instructions that generate two or more machine instructions. For instance, with some Z80 assemblers the instruction *ld hl,bc* is recognized to generate *ld l,c* followed by *ld h,b*.[4] These are sometimes known as *pseudo-opcodes*.

Mnemonics are arbitrary symbols; in 1985 the IEEE published Standard 694 for a uniform set of mnemonics to be used by all assemblers. The standard has since been withdrawn.

Data directives

There are instructions used to define data elements to hold data and variables. They define the type of data, the length and the alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined. Some assemblers classify these as pseudo-ops.

Assembly directives

Assembly directives, also called pseudo-opcodes, pseudo-operations or pseudo-ops, are instructions that are executed by an assembler at assembly time, not by a CPU at run time. The names of pseudo-ops often start with a dot to distinguish them from machine instructions. Pseudo-ops can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different applications. Or, a pseudo-op can be used to manipulate presentation of a program to make it easier to read and maintain. Another common use of pseudo-ops is to reserve storage areas for run-time data and optionally initialize their contents to known values.

Symbolic assemblers let programmers associate arbitrary names (*labels* or *symbols*) with memory locations and various constants. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting self-documenting code. In executable code, the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, GOTO destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Some assemblers, such as NASM provide flexible symbol management, letting programmers manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with relocatable addresses.

Assembly languages, like most other computer languages, allow comments to be added to program source code that will be ignored during assembly. Judicious commenting is essential in assembly language programs, as the meaning and purpose of a sequence of binary machine instructions can be difficult to determine. It should be noted that the "raw" (uncommented) assembly language generated by compilers or disassemblers is quite difficult to read when changes must be made.

Macros

Many assemblers support *predefined macros*, and others support *programmer-defined* (and repeatedly re-definable) macros involving sequences of text lines in which variables and constants are embedded. This sequence of text lines may include opcodes or directives. Once a macro has been defined its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, then processes them as if they existed in the source code file (including, in some assemblers, expansion of any macros existing in the replacement text).

Note that this definition of "macro" is slightly different from the use of the term in other contexts, like the C programming language. C macros created through the #define directive typically are just one line, or a few lines at most. Assembler macro instructions can be lengthy "programs" by themselves, executed by interpretation by the assembler during assembly.

Since macros can have 'short' names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be far shorter, requiring fewer lines of source code, as with higher level languages. They can also be used to add higher levels of structure to assembly programs, optionally introduce embedded debugging code via parameters and other similar features.

Macro assemblers often allow macros to take parameters. Some assemblers include quite sophisticated macro languages, incorporating such high-level language elements as optional parameters, symbolic variables, conditionals, string manipulation, and arithmetic operations, all usable during the execution of a given macro, and allowing macros to save context or exchange information. Thus a macro might generate numerous assembly language instructions or data definitions, based on the macro arguments. This could be used to generate record-style data structures or "unrolled" loops, for example, or could generate entire algorithms based on complex parameters. An organization using assembly language that has been heavily extended using such a macro suite can be considered to be working in a higher-level language, since such programmers are not working with a computer's lowest-level conceptual elements.

Macros were used to customize large scale software systems for specific customers in the mainframe era and were also used by customer personnel to satisfy their employers' needs by making specific versions of manufacturer operating systems. This was done, for example, by systems programmers working with IBM's Conversational Monitor System / Virtual Machine (VM/CMS) and with IBM's "real time transaction processing" add-ons, Customer Information Control System CICS, and ACP/TPF, the airline/financial system that began in the 1970s and still runs many large computer reservations systems (CRS) and credit card systems today.

It was also possible to use solely the macro processing abilities of an assembler to generate code written in completely different languages, for example, to generate a version of a program in COBOL using a pure macro assembler program containing lines of COBOL code inside assembly time operators instructing the assembler to generate arbitrary code.

This was because, as was realized in the 1960s, the concept of "macro processing" is independent of the concept of "assembly", the former being in modern terms more word processing, text processing, than generating object code. The concept of macro processing appeared, and appears, in the C programming language, which supports "preprocessor instructions" to set variables, and make conditional tests on their values. Note that unlike certain previous macro processors inside assemblers, the C preprocessor was not Turing-complete because it lacked the ability to either loop or "go to", the latter allowing programs to loop.

Despite the power of macro processing, it fell into disuse in many high level languages (major exceptions being C/C++ and PL/I) while remaining a perennial for assemblers.

Macro parameter substitution is strictly by name: at macro processing time, the value of a parameter is textually substituted for its name. The most famous class of bugs resulting was the use of a parameter that itself was an expression and not a simple name when the macro writer expected a name. In the macro: `foo: macro a load a*b` the intention was that the caller would provide the name of a variable, and the "global" variable or constant b would be used to multiply "a". If foo is called with the parameter `a-c`, the macro expansion of `load a-c*b` occurs. To avoid any possible ambiguity, users of macro processors can parenthesize formal parameters inside macro definitions, or callers can parenthesize the input parameters.

Support for structured programming

Some assemblers have incorporated structured programming elements to encode execution flow. The earliest example of this approach was in the Concept-14 macro set, originally proposed by Dr. H.D. Mills (March 1970), and implemented by Marvin Kessler at IBM's Federal Systems Division, which extended the S/360 macro assembler with IF/ELSE/ENDIF and similar control flow blocks. This was a way to reduce or eliminate the use of GOTO operations in assembly code, one of the main factors causing spaghetti code in assembly language. This approach was widely accepted in the early '80s (the latter days of large-scale assembly language use).

A curious design was A-natural, a "stream-oriented" assembler for 8080/Z80 processorsWikipedia:Citation needed from Whitesmiths Ltd. (developers of the Unix-like Idris operating system, and what was reported to be the first commercial C compiler). The language was classified as an assembler, because it worked with raw machine elements such as opcodes, registers, and memory references; but it incorporated an expression syntax to indicate

execution order. Parentheses and other special symbols, along with block-oriented structured programming constructs, controlled the sequence of the generated instructions. A-natural was built as the object language of a C compiler, rather than for hand-coding, but its logical syntax won some fans.

There has been little apparent demand for more sophisticated assemblers since the decline of large-scale assembly language development. In spite of that, they are still being developed and applied in cases where resource constraints or peculiarities in the target system's architecture prevent the effective use of higher-level languages.

Use of assembly language

Historical perspective

Assembly languages date to the introduction of the stored-program computer. The EDSAC computer (1949) had an assembler called *initial orders* featuring one-letter mnemonics. Nathaniel Rochester wrote an assembler for an IBM 701 (1954). SOAP (Symbolic Optimal Assembly Program) (1955) was an assembly language for the IBM 650 computer written by Stan Poley.

Assembly languages eliminated much of the error-prone and time-consuming first-generation programming needed with the earliest computers, freeing programmers from tedium such as remembering numeric codes and calculating addresses. They were once widely used for all sorts of programming. However, by the 1980s (1990s on microcomputers), their use had largely been supplanted by higher-level languages, in the search for improved programming productivity. Today assembly language is still used for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

Historically, numerous programs have been written entirely in assembly language. Operating systems were entirely written in assembly language until the introduction of the Burroughs MCP (1961), which was written in ESPOL, an Algol dialect. Many commercial applications were written in assembly language as well, including a large amount of the IBM mainframe software written by large corporations. COBOL, FORTRAN and some PL/I eventually displaced much of this work, although a number of large organizations retained assembly-language application infrastructures well into the '90s.

Most early microcomputers relied on hand-coded assembly language, including most operating systems and large applications. This was because these systems had severe resource constraints, imposed idiosyncratic memory and display architectures, and provided limited, buggy system services. Perhaps more important was the lack of first-class high-level language compilers suitable for microcomputer use. A psychological factor may have also played a role: the first generation of microcomputer programmers retained a hobbyist, "wires and pliers" attitude.

In a more commercial context, the biggest reasons for using assembly language were minimal bloat (size), minimal overhead, greater speed, and reliability.

Typical examples of large assembly language programs from this time are IBM PC DOS operating systems and early applications such as the spreadsheet program Lotus 1-2-3. Even into the 1990s, most console video games were written in assembly, including most games for the Mega Drive/Genesis and the Super Nintendo Entertainment System.Wikipedia:Citation needed According to some industry insiders, the assembly language was the best computer language to use to get the best performance out of the Sega Saturn, a console that was notoriously challenging to develop and program games for.[5] The popular arcade game NBA Jam (1993) is another example. Assembly language has long been the primary development language for many popular home computers of the 1980s and 1990s (such as the Sinclair ZX Spectrum, Commodore 64, Commodore Amiga, and Atari ST). This was in large part because BASIC dialects on these systems offered insufficient execution speed, as well as insufficient facilities to take full advantage of the available hardware on these systems. Some systems even have IDEs with highly advanced debugging and macro facilities.

The Assembler for the VIC-20 was written by Don French and published by *French Silk*. At 1,639 bytes in length, its author believes it is the smallest symbolic assembler ever written. The assembler supported the usual symbolic addressing and the definition of character strings or hex strings. It also allowed address expressions which could be combined with addition, subtraction, multiplication, division, logical AND, logical OR, and exponentiation operators.

Current usage

There have always been debates over the usefulness and performance of assembly language relative to high-level languages. Assembly language has specific niche uses where it is important; see below. Assembler can be used to optimize for speed or optimize for size. In the case of speed optimization, modern optimizing compilers are claimed to render high-level languages into code that can run as fast as hand-written assembly, despite the counter-examples that can be found. The complexity of modern processors and memory sub-systems makes effective optimization increasingly difficult for compilers, as well as assembly programmers. Moreover, and to the dismay of efficiency lovers, increasing processor performance has meant that most CPUs sit idle most of the time, with delays caused by predictable bottlenecks such as cache misses, I/O operations and paging. This has made raw code execution speed a non-issue for many programmers.

There are some situations in which developers might choose to use assembly language:

- A stand-alone executable of compact size is required that must execute without recourse to the run-time components or libraries associated with a high-level language; this is perhaps the most common situation. For example, firmware for telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors.
- Code that must interact directly with the hardware, for example in device drivers and interrupt handlers.
- Programs that need to use processor-specific instructions not implemented in a compiler. A common example is the bitwise rotation instruction at the core of many encryption algorithms.
- Programs that create vectorized functions for programs in higher-level languages such as C. In the higher-level language this is sometimes aided by compiler intrinsic functions which map directly to SIMD mnemonics, but nevertheless result in a one-to-one assembly conversion specific for the given vector processor.
- Programs requiring extreme optimization, for example an inner loop in a processor-intensive algorithm. Game programmers take advantage of the abilities of hardware features in systems, enabling games to run faster. Also large scientific simulations require highly optimized algorithms, e.g. linear algebra with BLAS or discrete cosine transformation (e.g. SIMD assembly version from x264)
- Situations where no high-level language exists, on a new or specialized processor, for example.
- Programs that need precise timing such as
  - real-time programs such as simulations, flight navigation systems, and medical equipment. For example, in a fly-by-wire system, telemetry must be interpreted and acted upon within strict time constraints. Such systems must eliminate sources of unpredictable delays, which may be created by (some) interpreted languages, automatic garbage collection, paging operations, or preemptive multitasking. However, some higher-level languages incorporate run-time components and operating system interfaces that can introduce such delays. Choosing assembly or lower-level languages for such systems gives programmers greater visibility and control over processing details.
  - cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.
- Situations where complete control over the environment is required, in extremely high security situations where nothing can be taken for granted.
- Computer viruses, bootloaders, certain device drivers, or other items very close to the hardware or low-level operating system.
- Instruction set simulators for monitoring, tracing and debugging where additional overhead is kept to a minimum

- Reverse-engineering and modifying program files such as
  - existing binaries that may or may not have originally been written in a high-level language, for example when trying to recreate programs for which source code is not available or has been lost, or cracking copy protection of proprietary software.
  - Video games (also termed ROM hacking), which is possible via several methods. The most widely employed is altering program code at the assembly language level.
- Self-modifying code, to which assembly language lends itself well.
- Games and other software for graphing calculators.

Assembly language is still taught in most computer science and electronic engineering programs. Although few programmers today regularly work with assembly language as a tool, the underlying concepts remain very important. Such fundamental topics as binary arithmetic, memory allocation, stack processing, character set encoding, interrupt processing, and compiler design would be hard to study in detail without a grasp of how a computer operates at the hardware level. Since a computer's behavior is fundamentally defined by its instruction set, the logical way to learn such concepts is to study an assembly language. Most modern computers have similar instruction sets. Therefore, studying a single assembly language is sufficient to learn: I) the basic concepts; II) to recognize situations where the use of assembly language might be appropriate; and III) to see how efficient executable code can be created from high-level languages. This is analogous to children needing to learn the basic arithmetic operations (e.g., long division), although calculators are widely used for all except the most trivial calculations.

Typical applications

- Assembly language is typically used in a system's boot code, (BIOS on IBM-compatible PC systems and CP/M), the low-level code that initializes and tests the system hardware prior to booting the operating system, and is often stored in ROM.
- Some compilers translate high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes.
- Relatively low-level languages, such as C, allow the programmer to embed assembly language directly in the source code. Programs using such facilities, such as the Linux kernel, can then construct abstractions using different assembly language on each hardware platform. The system's portable code can then use these processor-specific components through a uniform interface.
- Assembly language is valuable in reverse engineering. Many programs are distributed only in machine code form which is straightforward to translate into assembly language, but more difficult to translate into a higher-level language. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose.
- Assemblers can be used to generate blocks of data, with no high-level language overhead, from formatted and commented source code, to be used by other code.Wikipedia:Citation needed

Related terminology

- **Assembly language** or **assembler language** is commonly called **assembly**, **assembler**, **ASM**, or **symbolic machine code**. A generation of IBM mainframe programmers called it **ALC** for *Assembly Language Code* or **BAL**[6] for **Basic Assembly Language**. Calling the language **assembler** might be considered potentially confusing and ambiguous, since this is also the name of the utility program that translates assembly language statements into machine code. However, this usage has been common among professionals and in the literature for decades. Similarly, some early computers called their *assembler* their **assembly program**.
- The computational step where an assembler is run, including all macro processing, is termed **assembly time**. The assembler is said to be "assembling" the source code.
- The use of the word **assembly** dates from the early years of computers (*cf.* short code, speedcode).

- A **cross assembler** (see also cross compiler) is an assembler that is run on a computer or operating system of a different type from the system on which the resulting code is to run (the *target system*). Cross-assembling may be necessary if the target system cannot run an assembler itself, as is typically the case for small embedded systems. The computer on which the cross assembler is run must have some means of transporting the resulting machine code to the target system. Common methods involve transmitting an exact byte-by-byte copy of the machine code or an ASCII representation of the machine code in a portable format (such as Motorola or Intel hexadecimal) through a compatible interface to the target system for execution.
- An **assembler directive** or *pseudo-opcode* is a command given to an assembler "directing it to perform operations other than assembling instructions." Directives affect how the assembler operates and "may affect the object code, the symbol table, the listing file, and the values of internal assembler parameters." Sometimes the term *pseudo-opcode* is reserved for directives that generate object code, such as those that generate data.
- A **meta-assembler** is "a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language." [7]

List of assemblers for different computer architectures

Main article: List of assemblers

Further details

For any given personal computer, mainframe, embedded system, and game console, both past and present, at least one – possibly dozens – of assemblers have been written. For some examples, see the list of assemblers.

On Unix systems, the assembler is traditionally called as, although it is not a single body of code, being typically written anew for each port. A number of Unix variants use GAS.

Within processor groups, each assembler has its own dialect. Sometimes, some assemblers can read another assembler's dialect, for example, TASM can read old MASM code, but not the reverse. FASM and NASM have similar syntax, but each support different macros that could make them difficult to translate to each other. The basics are all the same, but the advanced features will differ.

Also, assembly can sometimes be portable across different operating systems on the same type of CPU. Calling conventions between operating systems often differ slightly or not at all, and with care it is possible to gain some portability in assembly language, usually by linking with a C library that does not change between operating systems.Wikipedia:Citation needed An instruction set simulator can process the object code/ binary of *any* assembler to achieve portability even across platforms with an overhead no greater than a typical bytecode interpreter.Wikipedia:Citation needed This is similar to use of microcode to achieve compatibility across a processor family.

Some higher level computer languages, such as C and Borland Pascal, support inline assembly where sections of assembly code, in practice usually brief, can be embedded into the high level language code. The Forth language commonly contains an assembler used in CODE words.

An emulator can be used to debug assembly-language programs.

Example listing of assembly language source code

The following is a partial listing generated by the NASM, an assembler for 32-bit Intel x86 CPUs. The code is for a subroutine, not a complete program.

```
                    ;------------------------------------------------------------
                    ; zstr_count:
                    ; Counts a zero-terminated ASCII string to determine its size
                    ; in:   eax = start address of the zero terminated string
                    ; out:  ecx = count = the length of the string

                    zstr_count:                 ; Entry point
00000030 B9FFFFFFFF     mov  ecx, -1            ; Init the loop counter, pre-decrement
                                                ;  to compensate for the increment
                    .loop:
00000035 41             inc  ecx                ; Add 1 to the loop counter
00000036 803C0800       cmp  byte [eax + ecx], 0  ; Compare the value at the string's
                                                ;  [starting memory address Plus the
                                                ;  loop offset], to zero
0000003A 75F9           jne  .loop              ; If the memory value is not zero,
                                                ;  then jump to the label called '.loop',
                                                ;  otherwise continue to the next line
                    .done:
                                                ; We don't do a final increment,
                                                ;  because even though the count is base 1,
                                                ;  we do not include the zero terminator in the
                                                ;  string's length
0000003C C3             ret                     ; Return to the calling program
```

The first column (from the left) is simply the line number in the listing and is otherwise meaningless. The second column is the relative address, in hex, of where the code will be placed in memory. The third column is the actual compiled code. For instance, `B9` is the x86 opcode for the `MOV ECX` instruction; `FFFFFFFF` is the value −1 in two's-complement binary form.

Names suffixed with colons (`:`) are symbolic labels; the labels do not create code, they are simply a way to tell the assembler that those locations have symbolic names. The `.done` label is only present for clarity of where the program ends, it does not serve any other purpose. Prefixing a period (`.`) on a label is a feature of the assembler, declaring the label as being local to the subroutine.

References

[1] Whether these bitgroups are orthogonal, or to what extent they are, depends on the CPU and instruction set design at hand.
[2] David Salomon (1993). *Assemblers and Loaders* (http://www.davidsalomon.name/assem.advertis/asl.pdf)
[3] Hyde, Randall. "Chapter 12 – Classes and Objects". The Art of Assembly Language, 2nd Edition. No Starch Press. © 2010.
[4] Z80 Op Codes for ZINT (http://www.z80.de/z80/z80code.htm). Z80.de. Retrieved on 2013-07-21.
[5] Eidolon's Inn : SegaBase Saturn (http://www.eidolons-inn.net/tiki-index.php?page=SegaBase+Saturn)
[6] Technically BAL was only the assembler for **BPS**; the others were macro assemblers.
[7] (John Daintith, ed.) A Dictionary of Computing: "meta-assembler" (http://www.encyclopedia.com/doc/1O11-metaassembler.html)

Further reading

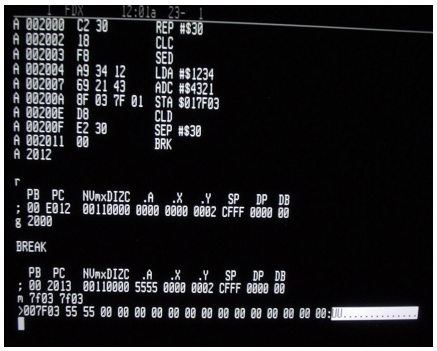
- Yurichev, Dennis, "An Introduction To Reverse Engineering for Beginners". Online book: http://yurichev.com/writings/RE_for_beginners-en.pdf
- *ASM Community Book* (http://www.asmcommunity.net/book/) "An online book full of helpful ASM info, tutorials and code examples" by the ASM Community
- Jonathan Bartlett: *Programming from the Ground Up* (http://programminggroundup.blogspot.com/). Bartlett Publishing, 2004. ISBN 0-9752838-4-7
  Also available online as PDF (http://download.savannah.gnu.org/releases-noredirect/pgubook/ProgrammingGroundUp-1-0-booksize.pdf)
- Robert Britton: *MIPS Assembly Language Programming*. Prentice Hall, 2003. ISBN 0-13-142044-5
- Paul Carter: *PC Assembly Language*. Free ebook, 2001.
  Website (http://drpaulcarter.com/pcasm/)
- Jeff Duntemann: *Assembly Language Step-by-Step*. Wiley, 2000. ISBN 0-471-37523-3
- Randall Hyde: *The Art of Assembly Language*. No Starch Press, 2003. ISBN 1-886411-97-2
  Draft versions available online (http://www.plantation-productions.com/Webster/www.artofasm.com/index.html) as PDF and HTML
- Peter Norton, John Socha, *Peter Norton's Assembly Language Book for the IBM PC*, Brady Books, NY: 1986.
- Michael Singer, *PDP-11. Assembler Language Programming and Machine Organization*, John Wiley & Sons, NY: 1980.
- Dominic Sweetman: *See MIPS Run*. Morgan Kaufmann Publishers, 1999. ISBN 1-55860-410-3
- John Waldron: *Introduction to RISC Assembly Language Programming*. Addison Wesley, 1998. ISBN 0-201-39828-1

External links

- Machine language for beginners (http://www.atariarchives.org/mlb/introduction.php)
- The ASM Community (http://www.asmcommunity.net/), a programming resource about assembly.
- Unix Assembly Language Programming (http://www.int80h.org/)
- Linux Assembly (http://asm.sourceforge.net/)
- IBM High Level Assembler (http://www-03.ibm.com/systems/z/os/zos/bkserv/r8pdf/index.html#hlasm)
  IBM manuals on mainframe assembler language.
- PPR: Learning Assembly Language (http://c2.com/cgi/wiki?LearningAssemblyLanguage)
- NASM - The Netwide Assembler (a popular assembly language) (http://www.nasm.us/)
- Assembly Language Programming Examples (http://www.azillionmonkeys.com/qed/asmexample.html)
- Authoring Windows Applications In Assembly Language (http://www.grc.com/smgassembly.htm)
- Iczelion's Win32 Assembly Tutorial (https://web.archive.org/web/*/http://win32assembly.online.fr/tutorials.html) at the Wayback Machine
- Assembly Optimization Tips (http://mark.masmcode.com/) by Mark Larson

Machine code

Machine code or **machine language** is a set of instructions executed directly by a computer's central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.



Machine language monitor in a W65C816S single-board computer, displaying code disassembly, as well as processor register and memory dumps.

Numerical machine code (i.e. not assembly code) may be regarded as the lowest-level representation of a compiled and/or assembled computer program or as a primitive and hardware-dependent programming language. While it is possible to write programs directly in numerical machine code, it is tedious and error prone to manage individual bits and calculate numerical addresses and constants manually. It is therefore rarely done today, except for situations that require extreme optimization or debugging.

Almost all practical programs today are written in higher-level languages or assembly language, and translated to executable machine code by a compiler and/or assembler and linker. Programs in interpreted languages[1] are not translated into machine code however, although their *interpreter* (which may be seen as an executor or processor) typically consists of directly executable machine code (generated from assembly and/or high level language source code).

Machine code instructions

Main article: Instruction set

Every processor or processor family has its own machine code instruction set. Instructions are patterns of bits that by physical design correspond to different commands to the machine. Thus, the instruction set is specific to a class of processors using (much) the same architecture. Successor or derivative processor designs often include all the instructions of a predecessor and may add additional instructions. Occasionally, a successor design will discontinue or alter the meaning of some instruction code (typically because it is needed for new purposes), affecting code compatibility to some extent; even nearly completely compatible processors may show slightly different behavior for some instructions, but this is rarely a problem. Systems may also differ in other details, such as memory arrangement, operating systems, or peripheral devices. Because a program normally relies on such factors, different systems will typically not run the same machine code, even when the same type of processor is used.

A machine code instruction set may have all instructions of the same length, or it may have variable-length instructions. How the patterns are organized varies strongly with the particular architecture and often also with the type of instruction. Most instructions have one or more opcode fields which specifies the basic instruction type (such as arithmetic, logical, jump, etc.) and the actual operation (such as add or compare) and other fields that may give the type of the operand(s), the addressing mode(s), the addressing offset(s) or index, or the actual value itself (such constant operands contained in an instruction are called *immediates*).[2]

Not all machines or individual instructions have explicit operands. An accumulator machine have a combined left operand and result in an implicit accumulator for most arithmetic instructions. Other architectures (such as 8086 and the x86-family) have accumulator versions of common instructions, with the accumulator regarded as one of the general registers by longer instructions. A stack machine has most or all of its operands on an implicit stack. Special purpose instructions also often lack explicit operands (CPUID in the x86 architecture writes values into four implicit

destination registers, for instance). This distinction between explicit and implicit operands is important in machine code generators, especially in the register allocation and live range tracking parts. A good code optimizer can track implicit as well as explicit operands which may allow more frequent constant propagation, constant folding of registers (a register assigned the result of a constant expression freed up by replacing it by that constant) and other code enhancements.

Programs

A computer program is a sequence of instructions that are executed by a CPU. While simple processors execute instructions one after another, superscalar processors are capable of executing several instructions at once.

Program flow may be influenced by special 'jump' instructions that transfer execution to an instruction other than the numerically following one. Conditional jumps are taken (execution continues at another address) or not (execution continues at the next instruction) depending on some condition.

Assembly languages

Main article: Assembly language

A much more readable rendition of machine language, called assembly language, uses mnemonic codes to refer to machine code instructions, rather than using the instructions' numeric values directly. For example, on the Zilog Z80 processor, the machine code `00000101`, which causes the CPU to decrement the `B` processor register, would be represented in assembly language as `DEC B`.

Example

The MIPS architecture provides a specific example for a machine code whose instructions are always 32 bits long. The general type of instruction is given by the *op* (operation) field, the highest 6 bits. J-type (jump) and I-type (immediate) instructions are fully specified by *op*. R-type (register) instructions include an additional field *funct* to determine the exact operation. The fields used in these types are:

```
   6      5     5     5     5      6 bits
[  op  |  rs |  rt |  rd |shamt| funct]  R-type
[  op  |  rs |  rt | address/immediate]  I-type
[  op  |        target address        ]  J-type
```

rs, *rt*, and *rd* indicate register operands; *shamt* gives a shift amount; and the *address* or *immediate* fields contain an operand directly.

For example adding the registers 1 and 2 and placing the result in register 6 is encoded:

```
[  op  |  rs |  rt |  rd |shamt| funct]
    0     1     2     6     0     32     decimal
 000000 00001 00010 00110 00000 100000   binary
```

Load a value into register 8, taken from the memory cell 68 cells after the location listed in register 3:

```
[  op  |  rs |  rt | address/immediate]
   35     3     8           68          decimal
 100011 00011 01000 00000 00001 000100   binary
```

Jumping to the address 1024:

```
[  op  |        target address        ]
    2                1024              decimal
```

```
000010 00000 00000 00000 10000 000000   binary
```

Relationship to microcode

In some computer architectures, the machine code is implemented by a more fundamental underlying layer of programs called microprograms, providing a common machine language interface across a line or family of different models of computer with widely different underlying dataflows. This is done to facilitate porting of machine language programs between different models. An example of this use is the IBM System/360 family of computers and their successors. With dataflow path widths of 8 bits to 64 bits and beyond, they nevertheless present a common architecture at the machine language level across the entire line.

Using a microcode layer to implement an emulator enables the computer to present the architecture of an entirely different computer. The System/360 line used this to allow porting programs from earlier IBM machines to the new family of computers, e.g. an IBM 1401/1440/1460 emulator on the IBM S/360 model 40.

Relationship to bytecode

Machine code should not be confused with so-called "bytecode" (or the older term p-code), which is either executed by an interpreter or itself compiled into machine code for faster (direct) execution. Machine code and assembly code is sometimes called *native code* when referring to platform-dependent parts of language features or libraries.

Storing in memory

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for the code (instructions) and data. Today, most processors implement such separate signal pathways for performance reasons but actually implement a Modified Harvard architecture,Wikipedia:Citation needed so they can support tasks like loading an executable program from disk storage as data and then executing it. Harvard architecture is contrasted to the Von Neumann architecture, where data and code are stored in the same memory.

From the point of view of a process, the *code space* is the part of its address space where the code in execution is stored. In multitasking systems this comprises the program's code segment and usually shared libraries. In multi-threading environment, different threads of one process share code space along with data space, which reduces the overhead of context switching considerably as compared to process switching.

Readability by humans

It has been said that machine code is so unreadable that the United States Copyright Office cannot identify whether a particular encoded program is an original work of authorship; however, the US Copyright Office *does* allow for copyright registration of computer programs. Hofstadter compares machine code with the genetic code: "Looking at a program written in machine language is vaguely comparable to looking at a DNA molecule atom by atom." (Note: first and third sources are from the early 1980s, and may be out of date.)

Notes and references

[1] Such as many versions of BASIC, especially early ones, as well as Smalltalk, MATLAB, Perl, Python, Ruby and other special purpose or scripting languages.

[2] Immediate operands (http://programmedlessons.org/AssemblyTutorial/Chapter-11/ass11_2.html)

Further reading

- Hennessy, John L.; Patterson, David A.. *Computer Organization and Design. The Hardware/Software Interface*. Morgan Kaufmann Publishers. ISBN 1-55860-281-X.
- Tanenbaum, Andrew S.. *Structured Computer Organization*. Prentice Hall. ISBN 0-13-020435-8.
- Brookshear, J. Glenn. *Computer Science: An Overview*. Addison Wesley. ISBN 0-321-38701-5.

Source code

For the 2011 film, see Source Code.

Not to be confused with source coding.

In computing, **source code** is any collection of computer instructions (possibly with comments) written using some human-readable computer language, usually as text. The source code of a program is specially designed to facilitate the work of computer programmers, who specify the actions to be performed by a computer mostly by writing source code. The source code is often transformed by a compiler program into low-level machine code understood by the computer. The machine code might then be stored for execution at a later time. Alternatively, an interpreter can be used to analyze and perform the outcomes of the source code program directly on the fly.

```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();
  frame.show();           // display the fra
}
```

An illustration of Java source code with prologue comments indicated in red, inline comments indicated in green, and program statements indicated in blue

Most computer applications are distributed in a form that includes executable files, but not their source code. If the source code were included, it would be useful to a user, programmer, or system administrator, who may wish to modify the program or understand how it works.

Aside from its machine-readable forms, source code also appears in books and other media; often in the form of small code snippets, but occasionally complete code bases; a well-known case is the source code of PGP.

Definitions

The notion of source code may also be taken more broadly, to include machine code and notations in graphical languages, neither of which are textual in nature. An example from an article presented on the annual IEEE conference on Source Code Analysis and Manipulation:[1]

> For the purpose of clarity '**source code**' is taken to mean any fully executable description of a software system. It is therefore so construed as to include machine code, very high level languages and executable graphical representations of systems.[2]

Organization

The source code which constitutes a program is usually held in one or more text files stored on a computer's hard disk; usually these files are carefully arranged into a directory tree, known as a **source tree**. Source code can also be stored in a database (as is common for stored procedures) or elsewhere.

The source code for a particular piece of software may be contained in a single file or many files. Though the practice is uncommon, a program's source code can be written in different programming languages.[3] For example, a program written primarily in the C programming language, might have portions written in assembly language for optimization purposes. It is also possible for some components of a piece of software to be written and compiled separately, in an arbitrary programming language, and later integrated into the software using a technique called library linking. This is the case in some languages, such as Java: each class is compiled separately into a file and linked by the interpreter at runtime.

Yet another method is to make the main program an interpreter for a programming languageWikipedia:Citation needed, either designed specifically for the application in question or general-purpose, and then write the bulk of the actual user functionality as macros or other forms of add-ins in this language, an approach taken for example by the GNU Emacs text editor.

The **code base** of a computer programming project is the larger collection of all the source code of all the computer programs which make up the project. It has become common practice to maintain code bases in version control systems. Moderately complex software customarily requires the compilation or assembly of several, sometimes dozens or even hundreds, of different source code files. In these cases, instructions for compilations, such as a Makefile, are included with the source code. These describe the relationships among the source code files, and contain information about how they are to be compiled.

The revision control system is another tool frequently used by developers for source code maintenance.

Purposes

Source code is primarily used as input to the process that produces an executable program (i.e., it is compiled or interpreted). It is also used as a method of communicating algorithms between people (e.g., code snippets in books).[4]

Programmers often find it helpful to review existing source code to learn about programming techniques. The sharing of source code between developers is frequently cited as a contributing factor to the maturation of their programming skills. Some people consider source code an expressive artistic medium.[5]

Porting software to other computer platforms is usually prohibitively difficult without source code. Without the source code for a particular piece of software, portability is generally computationally expensive.Wikipedia:Citation needed Possible porting options include binary translation and emulation of the original platform.

Decompilation of an executable program can be used to generate source code, either in assembly code or in a high-level language.

Programmers frequently adapt source code from one piece of software to use in other projects, a concept known as software reusability.

Licensing

Main article: Software license

Software, and its accompanying source code, typically falls within one of two licensing paradigms: open source and proprietary software.

Generally speaking, software is *open source* if the source code is free to use, distribute, modify and study, and *proprietary* if the source code is kept secret, or is privately owned and restricted. The first software license to be published and to explicitly grant these freedoms was the GNU General Public License in 1989. The GNU GPL was originally intended to be used with the GNU operating system.

For proprietary software, the provisions of the various copyright laws, trade secrecy and patents are used to keep the source code closed. Additionally, many pieces of retail software come with an end-user license agreement (EULA) which typically prohibits decompilation, reverse engineering, analysis, modification, or circumventing of copy protection. Types of source code protection – beyond traditional compilation to object code – include code encryption, code obfuscation or code morphing.

Legal issues in the United States

In a 2003 court case in the United States, it was ruled that source code should be considered a constitutionally protected form of free speech. Proponents of free speech argued that because source code conveys information to programmers, is written in a language, and can be used to share humour and other artistic pursuits, it is a protected form of communication.

One of the first court cases regarding the nature of source code as free speech involved University of California mathematics professor Dan Bernstein, who had published on the Internet the source code for an encryption program that he created. At the time, encryption algorithms were classified as munitions by the United States government; exporting encryption to other countries was considered an issue of national security, and had to be approved by the State Department. The Electronic Frontier Foundation sued the U.S. government on Bernstein's behalf; the court ruled that source code was free speech, protected by the First Amendment.

Quality

Main article: Software quality

The way a program is written can have important consequences for its maintainers. Coding conventions, which stress readability and some language-specific conventions, are aimed at the maintenance of the software source code, which involves debugging and updating. Other priorities, such as the speed of the program's execution, or the ability to compile the program for multiple architectures, often make code readability a less important consideration, since code *quality* generally depends on its *purpose*.

References

[1] SCAM Working Conference (http://www.ieee-scam.org/), 2001–2010.
[2] Why Source Code Analysis and Manipulation Will Always Be Important (http://www.cs.ucl.ac.uk/staff/M.Harman/scam10.pdf) by Mark Harman, 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010). Timişoara, Romania, 12–13 September 2010.
[3] Extending and Embedding the Python Interpreter — Python v2.6 Documentation (http://docs.python.org/extending/)
[4] Spinellis, D: *Code Reading: The Open Source Perspective*. Addison-Wesley Professional, 2003. ISBN 0-201-79940-5
[5] "*Art and Computer Programming*" ONLamp.com (http://www.onlamp.com/pub/a/onlamp/2005/06/30/artofprog.html), (2005)

- (VEW04) "Using a Decompiler for Real-World Source Recovery", M. Van Emmerik and T. Waddington, the *Working Conference on Reverse Engineering*, Delft, Netherlands, 9–12 November 2004. Extended version of the paper (http://www.itee.uq.edu.au/~emmerik/experience_long.pdf).

External links

- Source Code Definition (http://www.linfo.org/source_code.html) by The Linux Information Project (LINFO)
- "Obligatory accreditation system for IT security products (2008-09-22), may start from May 2009, reported by Yomiuri on 2009-04-24." (http://www.metafilter.com/75061/Obligatory-accreditation-system-for-IT-security-products). MetaFilter.com. Retrieved 2009-04-24.
- Same program written in multiple languages (http://rosettacode.org/wiki/Main_Page)

Command

"System command" redirects here. It is not to be confused with system call.

For other uses, see Command#Computing.

In computing, a **command** is a directive to a computer program acting as an interpreter of some kind, in order to perform a specific task. Most commonly a command is a directive to some kind of command-line interface, such as a shell.

Specifically, the term *command* is used in imperative computer languages. These languages are called this, because statements in these languages are usually written in a manner similar to the imperative mood used in many natural languages. If one views a statement in an imperative language as being like a sentence in a natural language, then a command is generally like a verb in such a language.

Many programs allow specially formatted arguments, known as flags or options, which modify the default behaviour of the command, while further arguments describe what the command acts on. Comparing to a natural language: the flags are adverbs, whilst the other arguments are objects.

Examples

Here are some commands given to a command-line interpreter (Unix shell).

The following command changes the user's place in the directory tree from their current position to the directory `/home/pete`. `cd` is the command and `/home/pete` is the argument:

```
cd /home/pete
```

The following command prints the text `Hello World` out to the standard output stream, which, in this case, will just print the text out on the screen. `echo` is the command and `"Hello World"` is the argument. The quotes are used to prevent `Hello` and `World` being treated as separate arguments:

```
echo "Hello World"
```

The following commands are equivalent. They list files in the directory `/bin`. `ls` is the command, `/bin` is the argument and there are three flags: `-l`, `-t` and `-r`:

```
ls -l -t -r  /lvl 100
ls -ltr  /bin
```

The following command displays the contents of the files `ch1.txt` and `ch2.txt`. `cat` is the command and `ch1.txt` and `ch2.txt` are both arguments.

```
cat ch1.txt ch2.txt
```

The following command lists all the contents of the current directory. `dir` is the command and "A" is a flag. There is no argument. Here are some commands given to a different command-line interpreter (the DOS, OS/2 and Microsoft Windows command prompt). Notice that the flags are identified differently but that the concepts are the same:

```
dir /A
```

The following command displays the contents of the file readme.txt. `type` is the command. "readme.txt" is the argument. "P" is a parameter...

```
type /P readme.txt
```

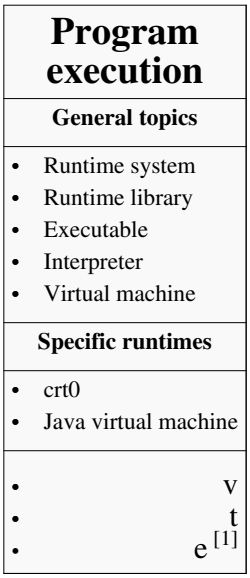
External links

- command [1] from FOLDOC

References

[1] http://foldoc.org/index.cgi?query=command

Execution



Execution in computer and software engineering is the process by which a computer or a virtual machine performs the instructions of a computer program. The instructions in the program trigger sequences of simple actions on the executing machine. Those actions produce effects according to the semantics of the instructions in the program.

Programs for a computer may execute in a batch process without human interaction, or a user may type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together.

The term **run** is used almost synonymously. A related meaning of both "to run" and "to execute" refers to the specific action of a user starting (or *launching* or *invoking*) a program, as in "Please run the ... application."

Context of execution

The context in which execution takes place is crucial. Very few programs execute on a bare machine. Programs usually contain implicit and explicit assumptions about resources available at the time of execution. Most programs execute with the support of an operating system and run-time libraries specific to the source language that provide crucial services not supplied directly by the computer itself. This supportive environment, for instance, usually decouples a program from direct manipulation of the computer peripherals, providing more general, abstract services instead.

Interpreter

A system that executes a program is called an interpreter of the program. Loosely speaking, an interpreter actually does what the program says to do. This contrasts with a language translator that converts a program from one language to another. The most common language translators are compilers. Translators typically convert their source from a high-level, human readable language into a lower-level language (sometimes as low as native machine code) that is simpler and faster for the processor to directly execute. The ideal is that the ratio of executions to translations of a program will be large; that is, a program need only be compiled once and can be run any number of times. This can provide a large benefit for translation versus direct interpretation of the source language. One trade-off is that development time is increased, because of the compilation. In some cases, only the changed files must be recompiled. Then the executable needs to be relinked. For some changes, the executable must be rebuilt from scratch. As computers and compilers become faster, this fact becomes less of an obstacle. Also, the speed of the end

product is typically more important to the user than the development time.

Translators usually produce an abstract result that is not completely ready to execute. Frequently, the operating system will convert the translator's object code into the final executable form just before execution of the program begins. This usually involves modifying the code to bind it to real hardware addresses and establishing address links between the program and support code in libraries. In some cases this code is further transformed the first time it is executed, for instance by just-in-time compilers, into a more efficient form that persists for some period, usually at least during the current execution run.

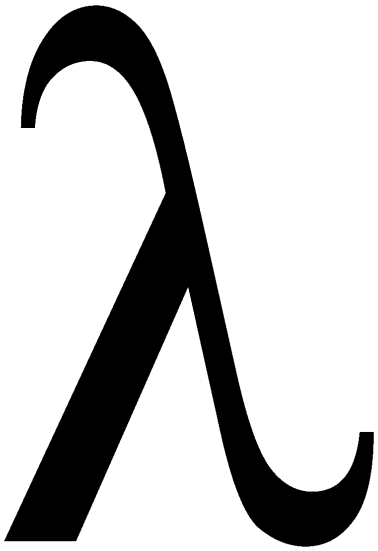
References

[1] http://en.wikipedia.org/w/index.php?title=Template:Program_execution&action=edit

Theory

Programming language theory

Programming language theory (**PLT**) is a branch of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features. It falls within the discipline of computer science, both depending on and affecting mathematics, software engineering and linguistics. It is a well-recognized branch of computer science, and an active research area, with results published in numerous journals dedicated to PLT, as well as in general computer science and engineering publications.



The lowercase Greek letter λ (lambda) is an unofficial symbol of the field of programming language theory. This usage derives from the lambda calculus, a computational model introduced by Alonzo Church in the 1930s and widely used by programming language researchers. It graces the cover of the classic text *Structure and Interpretation of Computer Programs*, and the title of the so-called Lambda Papers, written by Gerald Jay Sussman and Guy Steele, the developers of the Scheme programming language.

History

In some ways, the history of programming language theory predates even the development of programming languages themselves. The lambda calculus, developed by Alonzo Church and Stephen Cole Kleene in the 1930s, is considered by some to be the world's first programming language, even though it was intended to *model* computation rather than being a means for programmers to *describe* algorithms to a computer system. Many modern functional programming languages have been described as providing a "thin veneer" over the lambda calculus,[1] and many are easily described in terms of it.

The first programming language to be proposed was Plankalkül, which was designed by Konrad Zuse in the 1940s, but not publicly known until 1972 (and not implemented until 1998). The first widely known and successful programming language was Fortran, developed from 1954 to 1957 by a team of IBM researchers led by John Backus. The success of FORTRAN led to the formation of a committee of scientists to develop a "universal" computer language; the result of their effort was ALGOL 58. Separately, John McCarthy of MIT developed the Lisp programming language (based on the lambda calculus), the first language with origins in academia to be successful. With the success of these initial efforts, programming languages became an active topic of research in the 1960s and beyond.

Some other key events in the history of programming language theory since then:

1950s

- Noam Chomsky developed the Chomsky hierarchy in the field of linguistics; a discovery which has directly impacted programming language theory and other branches of computer science.

1960s

- The Simula language was developed by Ole-Johan Dahl and Kristen Nygaard; it is widely considered to be the first example of an object-oriented programming language; Simula also introduced the concept of coroutines.
- In 1964, Peter Landin is the first to realize Church's lambda calculus can be used to model programming languages. He introduces the SECD machine which "interprets" lambda expressions.
- In 1965, Landin introduces the J operator, essentially a form of continuation.
- In 1966, Landin introduces ISWIM, an abstract computer programming language in his article *The Next 700 Programming Languages*. It is influential in the design of languages leading to the Haskell programming language.
- In 1966, Corrado Böhm introduced the programming language CUCH (Curry-Church).[2]
- In 1967, Christopher Strachey publishes his influential set of lecture notes *Fundamental Concepts in Programming Languages*, introducing the terminology *R-values*, *L-values*, *parametric polymorphism*, and *ad hoc polymorphism*.
- In 1969, J. Roger Hindley publishes *The Principal Type-Scheme of an Object in Combinatory Logic*, later generalized into the Hindley–Milner type inference algorithm.
- In 1969, Tony Hoare introduces the Hoare logic, a form of axiomatic semantics.
- In 1969, William Alvin Howard observed that a "high-level" proof system, referred to as natural deduction, can be directly interpreted in its intuitionistic version as a typed variant of the model of computation known as lambda calculus. This became known as the Curry–Howard correspondence.

1970s

- In 1970, Dana Scott first publishes his work on denotational semantics.
- In 1972, Logic programming and Prolog were developed thus allowing computer programs to be expressed as mathematical logic.
- In 1974, John C. Reynolds discovers System F. It had already been discovered in 1971 by the mathematical logician Jean-Yves Girard.
- From 1975, Sussman and Steele develop the Scheme programming language, a Lisp dialect incorporating lexical scoping, a unified namespace, and elements from the Actor model including first-class continuations.
- Backus, at the 1977 ACM Turing Award lecture, assailed the current state of industrial languages and proposed a new class of programming languages now known as function-level programming languages.
- In 1977, Gordon Plotkin introduces Programming Computable Functions, an abstract typed functional language.
- In 1978, Robin Milner introduces the Hindley–Milner type inference algorithm for the ML programming language. Type theory became applied as a discipline to programming languages, this application has led to tremendous advances in type theory over the years.

1980s

- In 1981, Gordon Plotkin publishes his paper on structured operational semantics.
- In 1988, Gilles Kahn published his paper on natural semantics.
- A team of scientists at Xerox PARC led by Alan Kay develop Smalltalk, an object-oriented language widely known for its innovative development environment.
- There emerged process calculi, such as the Calculus of Communicating Systems of Robin Milner, and the Communicating sequential processes model of C. A. R. Hoare, as well as similar models of concurrency such as the Actor model of Carl Hewitt.
- In 1985, The release of Miranda sparks an academic interest in lazy-evaluated pure functional programming languages. A committee was formed to define an open standard resulting in the release of the Haskell 1.0 standard in 1990.

- Bertrand Meyer created the methodology Design by contract and incorporated it into the Eiffel programming language.

1990s

- Gregor Kiczales, Jim Des Rivieres and Daniel G. Bobrow published the book The Art of the Metaobject Protocol.
- Eugenio Moggi and Philip Wadler introduced the use of monads for structuring programs written in functional programming languages.

Sub-disciplines and related fields

There are several fields of study which either lie within programming language theory, or which have a profound influence on it; many of these have considerable overlap. In addition, PLT makes use of many other branches of mathematics, including computability theory, category theory, and set theory.

Formal semantics

Main article: Formal semantics of programming languages

Formal semantics is the formal specification of the behaviour of computer programs and programming languages. Three common approaches to describe the semantics or "meaning" of a computer program are denotational semantics, operational semantics and axiomatic semantics.

Type theory

Main article: type theory

Type theory is the study of type systems; which are "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".[3] Many programming languages are distinguished by the characteristics of their type systems.

Program analysis and transformation

Main articles: Program analysis and Program transformation

Program analysis is the general problem of examining a program and determining key characteristics (such as the absence of classes of program errors). Program transformation is the process of transforming a program in one form (language) to another form.

Comparative programming language analysis

Comparative programming language analysis seeks to classify programming languages into different types based on their characteristics; broad categories of programming languages are often known as programming paradigms.

Generic and metaprogramming

Metaprogramming is the generation of higher-order programs which, when executed, produce programs (possibly in a different language, or in a subset of the original language) as a result.

Domain-specific languages

Domain-specific languages are languages constructed to efficiently solve problems in a particular problem domain.

Compiler construction

Main article: Compiler construction

Compiler theory is the theory of writing *compilers* (or more generally, *translators*); programs which translate a program written in one language into another form. The actions of a compiler are traditionally broken up into *syntax analysis* (scanning and parsing), *semantic analysis* (determining what a program should do), *optimization* (improving the performance of a program as indicated by some metric; typically execution speed) and *code generation* (generation and output of an equivalent program in some target language; often the instruction set of a CPU).

Run-time systems

Runtime systems refers to the development of programming language runtime environments and their components, including virtual machines, garbage collection, and foreign function interfaces.

Journals, publications, and conferences

Conferences are the primary venue for presenting research in programming languages. The most well known conferences include the Symposium on Principles of Programming Languages (POPL), Conference on Programming Language Design and Implementation (PLDI), the International Conference on Functional Programming (ICFP), and the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA).

Notable journals that publish PLT research include the *ACM Transactions on Programming Languages and Systems* (TOPLAS), *Journal of Functional Programming* (JFP), *Journal of Functional and Logic Programming*, and *Higher-Order and Symbolic Computation*.

References

[1] http://www.c2.com/cgi/wiki?ModelsOfComputation

[2] C. Böhm and W. Gross (1996). Introduction to the CUCH. In E. R. Caianiello (ed.), *Automata Theory*, p. 35-64/

[3] Benjamin C. Pierce. 2002. Types and Programming Languages. MIT Press, Cambridge, MA, USA.

Further reading

See also: Programming language § Further reading and Semantics of programming languages § Further reading

- Abadi, Martín and Cardelli, Luca. *A Theory of Objects*. Springer-Verlag.
- Michael J. C. Gordon. *Programming Language Theory and Its Implementation*. Prentice Hall.
- Gunter, Carl and Mitchell, John C. (eds.). *Theoretical Aspects of Object Oriented Programming Languages: Types, Semantics, and Language Design*. MIT Press.
- Harper, Robert. *Practical Foundations for Programming Languages* (http://www.cs.cmu.edu/~rwh/plbook/book.pdf). Draft version.
- Knuth, Donald E. (2003). *Selected Papers on Computer Languages* (http://www-cs-faculty.stanford.edu/~uno/cl.html). Stanford, California: Center for the Study of Language and Information.
- Mitchell, John C.. *Foundations for Programming Languages*.
- Mitchell, John C.. *Introduction to Programming Language Theory*.

- O'Hearn, Peter. W. and Tennent, Robert. D. (1997). *Algol-like Languages* (http://www.eecs.qmul.ac.uk/~ohearn/Algol/algol.html). Progress in Theoretical Computer Science. Birkhauser, Boston.
- Pierce, Benjamin C. (2002). *Types and Programming Languages* (http://www.cis.upenn.edu/~bcpierce/tapl/main.html). MIT Press.
- Pierce, Benjamin C. *Advanced Topics in Types and Programming Languages*.
- Pierce, Benjamin C. *et al.* (2010). *Software Foundations* (http://www.cis.upenn.edu/~bcpierce/sf/).

External links

- Lambda the Ultimate (http://lambda-the-ultimate.org/policies#Purpose), a community weblog for professional discussion and repository of documents on programming language theory.
- Great Works in Programming Languages (http://www.cis.upenn.edu/~bcpierce/courses/670Fall04/GreatWorksInPL.shtml). Collected by Benjamin C. Pierce (University of Pennsylvania).
- Classic Papers in Programming Languages and Logic (http://www.cs.cmu.edu/~crary/819-f09/). Collected by Karl Crary (Carnegie Mellon University).
- Programming Language Research (http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/web/language-research.html). Directory by Mark Leone.
- Programming Language Theory Texts Online (http://www.cs.uu.nl/wiki/Techno/ProgrammingLanguageTheoryTextsOnline). At Utrecht University.
- λ-Calculus: Then & Now (http://turing100.acm.org/lambda_calculus_timeline.pdf) by Dana S. Scott for the ACM Turing Centenary Celebration
- Grand Challenges in Programming Languages (http://plgrand.blogspot.com/). Panel session at POPL 2009.

Type system

This article is about type systems from the point-of-view of computer programming. For a theoretical formulation, see type theory.

Type systems
• Type safety • Dynamic type-checking • Static type-checking • Inferred vs. Manifest • Nominal vs. Structural • Dependent typing • Duck typing • Latent typing • Substructural typing • Uniqueness typing • Strong and weak typing
• v • t • e [1]

In programming languages, a **type system** is a collection of rules that assign a property called a *type* to the various constructs—such as variables, expressions, functions or modules—a computer program is composed of.[2] The main purpose of a type system is to reduce bugs in computer programs[3] by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can

happen statically (at compile time), dynamically (at run time), or it can happen as a combination of static and dynamic checking. Type systems have other purposes as well, such as enabling certain compiler optimizations, allowing for multiple dispatch, providing a form of documentation, etc.

An example of a simple type system is that of the C language. The portions of a C program are the function definitions. One function is invoked by another function. The interface of a function states the name of the function and a list of values that are passed to the function's code. The code of an invoking function states the name of the invoked, along with the names of variables that hold values to pass to it. During execution, the values are placed into temporary storage, then execution jumps to the code of the invoked function. The invoked function's code accesses the values and makes use of them. If the instructions inside the function are written with the assumption of receiving an integer value, but the calling code passed a floating-point value, then the wrong result will be computed by the invoked function. The C compiler checks the type declared for each variable sent, against the type declared for each variable in the interface of the invoked function. If the types do not match, the compiler throws a compile-time error.

In greater technical depth, a type-system associates a *type* with each computed value. By examining the flow of these values, a type system attempts to ensure or prove that no *type errors* can occur. The particular type system in question determines exactly what constitutes a type error, but in general the aim is to prevent operations expecting a certain kind of value from being used with values for which that operation does not make sense (logic errors); memory errors will also be prevented. Type systems are often specified as part of programming languages, and built into the interpreters and compilers for them; although the type system of a language can be extended by optional tools that perform additional kinds of checks using the language's original type syntax and grammar.

A compiler may also use the static type of a value to optimize the storage it needs and the choice of algorithms for operations on the value. In many C compilers the *float* data type, for example, is represented in 32 bits, in accord with the IEEE specification for single-precision floating point numbers. They will thus use floating-point-specific microprocessor operations on those values (floating-point addition, multiplication, etc.).

The depth of type constraints and the manner of their evaluation affect the *typing* of the language. A programming language may further associate an operation with varying concrete algorithms on each type in the case of type polymorphism. Type theory is the study of type systems, although the concrete type systems of programming languages originate from practical issues of computer architecture, compiler implementation, and language design.

Fundamentals

Formally, type theory studies type systems. A programming language must have occurrence to type check using the *type system* whether at compiler time or runtime, manually annotated or automatically inferred. As Mark Manasse concisely put it:[4]

> The fundamental problem addressed by a type theory is to ensure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.

Assigning a data type, what is called *typing*, gives meaning to a sequences of bits such as a value in memory or some object such as a variable. The hardware of a general purpose computer is unable to discriminate between for example a memory address and an instruction code, or between a character, an integer, or a floating-point number, because it makes no intrinsic distinction between any of the possible values that a sequence of bits might *mean*. Associating a sequence of bits with a type conveys that meaning to the programmable hardware to form a *symbolic system* composed of that hardware and some program.

A program associates each value with at least one particular type, but it also can occur that one value is associated with many subtypes. Other entities, such as objects, modules, communication channels, dependencies can become associated with a type. Even a type can become associated with a type. An implementation of some *type system* could in theory associate some identifications named this way:

- data type – a type of a value
- class – a type of an object
- kind (type theory) – a *type of a type*, or metatype

These are the kinds of abstractions typing can go through on a hierarchy of levels contained in a system.

When a programming language evolves a more elaborate type system, it gains a more finely-grained rule set than basic type checking, but this comes at a price when the type inferences (and other properties) become undecidable, and when more attention must be paid by the programmer to annotate code or to consider computer-related operations and functioning. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in a type safe manner.

The more type restrictions that are imposed by the compiler, the more *strongly typed* a programming language is. Strongly typed languages often require the programmer to make explicit conversions in contexts where an implicit conversion would cause no harm. Pascal's type system has been described as "too strong" because, for example, the size of an array or string is part of its type, making some programming tasks difficult.[5][6] Haskell is also strongly typed but its types are automatically inferred so that explicit conversions are unnecessary.

A programming language compiler can also implement a *dependent type* or an *effect system*, which enables even more program specifications to be verified by a type checker. Beyond simple value-type pairs, a virtual "region" of code is associated with an "effect" component describing *what* is being done *with what*, and enabling for example to "throw" an error report. Thus the symbolic system may be a *type and effect system*, which endows it with more safety checking than type checking alone.

Whether automated by the compiler or specified by a programmer, a type system makes program behavior illegal that is outside the type-system rules. Advantages provided by programmer-specified type systems include:

- *Abstraction* (or *modularity*) – Types enable programmers to think at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can begin to think of a string as a collection of character values instead of as a mere array of bytes. Higher still, types enable programmers to think about and express interfaces between two of *any*-sized subsystems. This enables more levels of localization so that the definitions required for interoperability of the subsystems remain consistent when those two subsystems communicate.
- *Documentation* – In more expressive type systems, types can serve as a form of documentation clarifying the intent of the programmer. For instance, if a programmer declares a function as returning a timestamp type, this documents the function when the timestamp type can be explicitly declared deeper in the code to be integer type.

Advantages provided by compiler-specified type systems include:

- *Optimization* – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions.
- *Safety* – A type system enables the compiler to detect meaningless or probably invalid code. For example, we can identify an expression `3 / "Hello, World"` as invalid, when the rules do not specify how to divide an integer by a string. Strong typing offers more safety, but cannot guarantee complete *type safety*.

Type safety contributes to program correctness, but can only guarantee correctness at the expense of making the type checking itself an undecidable problem. In a *type system* with automated type checking a program may prove to run incorrectly yet be safely typed, and produce no compiler errors. Division by zero is an unsafe and incorrect operation, but a type checker running only at compile time doesn't scan for division by zero in most programming languages, and then it is left as a runtime error. To prove the absence of these more-general-than-types defects, other kinds of formal methods, collectively known as program analyses, are in common use. In addition software testing is an empirical method for finding errors that the type checker cannot detect.

Type checking

The process of verifying and enforcing the constraints of types – *type checking* – may occur either at compile-time (a static check) or run-time (a dynamic check). If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as *strongly typed*, if not, as *weakly typed*. The terms are not usually used in a strict sense.

Static type-checking

Static type-checking is the process of verifying the type safety of a program based on analysis of a program's text (source code). If a program passes a static type-checker, then the program is guaranteed to satisfy some set of type-safety properties for all possible inputs.

Because static type-checking operates on a program's text, it allows many bugs to be caught early in the development cycle.

Static type-checking can be thought of as a limited form of program verification (see type safety). In a type-safe language, static type-checking can also be thought of as an optimization. If a compiler can prove that a program is well-typed, then it does not need to emit dynamic safety checks, allowing the resulting compiled binary to run faster.

Static type-checking for Turing-complete languages is inherently conservative. That is, if a type system is both *sound* (meaning that it rejects all incorrect programs) and *decidable* (meaning that it is possible to write an algorithm which determines whether a program is well-typed), then it will always be possible to define a program which is well-typed but which does not satisfy the type-checker. For example, consider a program containing the code:

```
if <complex test> then <do something> else <generate type error>
```

Even if the expression `<complex test>` always evaluates to `true` at run-time, most type-checkers will reject the program as ill-typed, because it is difficult (if not impossible) for a static analyzer to determine that the `else` branch will not be taken.[7] Conversely, a static type-checker will quickly detect type errors in rarely-used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors, because the combination of all places where values are created and all places where a certain value is used must be taken into account.

A number of useful and common programming language features cannot be checked statically, such as downcasting. Therefore, many languages will have both static and dynamic type-checking; the static type-checker verifies what it can, and dynamic checks verify the rest.

Many languages with static type-checking provide a way to bypass the type checker. Some languages allow programmers to choose between static and dynamic type safety. For example, C# distinguishes between "statically-typed" and "dynamically-typed" variables; uses of the former are checked statically, while uses of the latter are checked dynamically. Other languages allow users to write code which is not type-safe. For example, in C, programmers can freely cast a value between any two types which have the same size.

For a list of languages with static type-checking, see the category for statically typed languages.

Dynamic type-checking and runtime type information

Dynamic type-checking is the process of verifying the type safety of a program at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a "type tag" (i.e., a reference to a type) containing its type information. This runtime type information (RTTI) can also be used to implement dynamic dispatch, late binding, downcasting, reflection, and similar features.

Most type-safe languages include some form of dynamic type-checking, even if they also have a static type checker. The reason for this is that many useful features or properties are difficult or impossible to verify statically. For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, then the operation is legal only if the value being converted is actually a value of type B. Therefore, a dynamic check is needed to verify that the operation is safe.

By definition, dynamic type-checking may cause a program to fail at runtime. In some programming languages, it is possible to anticipate and recover from these failures. In others, type-checking errors are considered fatal.

Programming languages which include dynamic type-checking but not static type-checking are often called "dynamically-typed programming languages". For a list of such languages, see the category for dynamically typed programming languages.

Combining static and dynamic type-checking

The presence of static type-checking in a programming language does not necessarily imply the absence of dynamic type-checking. For example, Java and some other ostensibly statically typed languages support downcasting types to their subtypes, querying an object to discover its dynamic type and other type operations that depend on runtime type information. More generally, most programming languages include mechanisms for dispatching over different 'kinds' of data, such as disjoint unions, subtype polymorphism, and variant types. Even when not interacting with type annotations or type checking, such mechanisms are materially similar to dynamic typing implementations. See programming language for more discussion of the interactions between static and dynamic typing.

Objects in object oriented languages are usually accessed by a reference whose static target type (or manifest type) is equal to either the object's run-time type (its latent type) or a supertype thereof. This is conformant with the Liskov substitution principle that states that all operations performed on an instance of a given type can also be performed on an instance of a subtype. This concept is also known as subsumption. In some languages subtypes may also possess covariant or contravariant return types and argument types respectively.

Certain languages, for example Clojure, Common Lisp, or Cython, are dynamically type-checked by default, but allow programs to opt into static type-checking by providing optional annotations. One reason to use such hints would be to optimize the performance of critical sections of a program.

As of version 4.0, the C# language provides a way to indicate that a variable should not be statically type-checked. A variable whose type is `dynamic` will not be subject to static type-checking. Instead, the program relies on runtime type information to determine how the variable may be used.

Static and dynamic type checking in practice

The choice between static and dynamic typing requires trade-offs.

Static typing can find type errors reliably at compile time. This should increase the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, and thus disagree over the proportion of those bugs that are coded that would be caught by appropriately representing the designed types in code. Static typing advocates believe programs are more reliable when they have been well type-checked, while dynamic typing advocates point to distributed code that has proven reliable and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased. Advocates of dependently typed languages such as Dependent ML and Epigram have suggested that almost all bugs can be considered type

errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.

Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce optimized machine code. Further, compilers for statically typed languages can find assembler shortcuts more easily. Some dynamically typed languages such as Common Lisp allow optional type declarations for optimization for this very reason. Static typing makes this pervasive. See optimization.

By contrast, dynamic typing may allow compilers to run more quickly and allow interpreters to dynamically load new code, since changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit. This too may reduce the edit-compile-test-debug cycle.

Statically typed languages that lack type inference (such as C and Java) require that programmers declare the types they intend a method or function to use. This can serve as additional documentation for the program, which the compiler will not permit the programmer to ignore or permit to drift out of synchronization. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala, OCaml, F# and to a lesser extent C#), so explicit type declaration is not a necessary requirement for static typing in all languages.

Dynamic typing allows constructs that some static type checking would reject as illegal. For example, *eval* functions, which execute arbitrary data as code, become possible. An *eval* function is possible with static typing, but requires advanced uses of algebraic data types. Furthermore, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full-fledged data structure (usually for the purposes of experimentation and testing).

Dynamic typing typically allows duck typing (which enables easier code reuse). Many languages with static typing also feature duck typing or other mechanisms like generic programming which also enables easier code reuse.

Dynamic typing typically makes metaprogramming easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code. More advanced run-time constructs such as metaclasses and introspection are often more difficult to use in statically typed languages. In some languages, such features may also be used e.g. to generate new types and behaviors on the fly, based on run-time data. Such advanced constructs are often provided by dynamic programming languages; many of these are dynamically typed, although *dynamic typing* need not be related to *dynamic programming languages*.

"Strong" and "weak" type systems

Main article: Strong and weak typing

Languages are often colloquially referred to as "strongly typed" or "weakly typed". In fact, there is no universally accepted definition of what these terms mean. In general, there are more precise terms to represent the differences between type systems that lead people to call them "strong" or "weak".

Type safety and memory safety

Main article: Type safety

A third way of categorizing the type system of a programming language uses the safety of typed operations and conversions. Computer scientists consider a language "type-safe" if it does not allow operations or conversions that violate the rules of the type system.

Some observers use the term *memory-safe language* (or just *safe language*) to describe languages that do not allow programs to access memory that has not been assigned for their use. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors.

Consider the following program of a language that is both type-safe and memory-safe:[8]

```
var x := 5;
var y := "37";
var z := x + y;
```

In this example, the variable `z` will have the value 42. While this may not be what the programmer anticipated, it is a well-defined result. If `y` was a different string, one that could not be converted to a number (e.g. "Hello World"), the result would be well-defined as well. Note that a program can be type-safe or memory-safe and still crash on an invalid operation; in fact, if a program encounters an operation which is not type-safe, terminating the program is often the only option.

Now consider a similar example in C:

```
int x = 5;
char y[] = "37";
char* z = x + y;
```

In this example `z` will point to a memory address five characters beyond `y`, equivalent to three characters after the terminating zero character of the string pointed to by `y`. This is memory that the program is not expected to access. It may contain garbage data, and it certainly doesn't contain anything useful. As this example shows, C is neither a memory-safe nor a type-safe language.

In general, type-safety and memory-safety go hand in hand. For example, a language which supports pointer arithmetic and number-to-pointer conversions (like C) is neither memory-safe nor type-safe, since it allows arbitrary memory to be accessed as if it were valid memory of any type.

For more information, see memory safety.

Variable levels of type checking

Some languages allow different levels of checking to apply to different regions of code. Examples include:-

- The `use strict` directive in javascript[9][10][11] and Perl applies stronger checking.
- The `@` operator in PHP suppresses some error messages.
- The `Option Strict On` in VB.NET allows the compiler to require a conversion between objects.

Additional tools such as lint and IBM Rational Purify can also be used to achieve a higher level of strictness.

Optional type systems

It has been proposed, chiefly by Gilad Bracha, that the choice of type system be made independent of choice of language; that a type system should be a module that can be "plugged" into a language as required. He believes this is advantageous, because what he calls mandatory type systems make languages less expressive and code more fragile.[12] The requirement that types do not affect the semantics of the language is difficult to fulfill; for instance, class-based inheritance becomes impossible.Wikipedia:Citation needed

Optional typing is related to gradual typing, but still distinct from it.[13]WP:NOTRS

Polymorphism and types

Main article: Polymorphism (computer science)

The term "polymorphism" refers to the ability of code (in particular, methods or classes) to act on values of multiple types, or to the ability of different instances of the same data structure to contain elements of different types. Type systems that allow polymorphism generally do so in order to improve the potential for code re-use: in a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once, rather than once for each type of element with which they plan to use it. For this reason computer scientists sometimes call the use of certain forms of polymorphism *generic programming*. The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping.

Duck typing

Main article: Duck typing

In "duck typing", a statement calling a method `m` on an object does not rely on the declared type of the object; only that the object, of whatever type, must supply an implementation of the method called, when called, at run-time.

Duck typing differs from structural typing in that, if the "part" (of the whole module structure) needed for a given local computation is present *at runtime*, the duck type system is satisfied in its type identity analysis. On the other hand, a structural type system would require the analysis of the whole module structure at compile time to determine type identity or type dependence.

Duck typing differs from a nominative type system in a number of aspects. The most prominent ones are that for duck typing, type information is determined at runtime (as contrasted to compile time), and the name of the type is irrelevant to determine type identity or type dependence; only partial structure information is required for that for a given point in the program execution.

Duck typing uses the premise that (referring to a value) "if it walks like a duck, and quacks like a duck, then it is a duck" (this is a reference to the duck test that is attributed to James Whitcomb Riley). The term may have been coined Wikipedia:Citation needed by Alex Martelli in a 2000 message to the comp.lang.python newsgroup (see Python).

Duck typing has been demonstrated to increase programmer productivity in a controlled experiment.[14]Wikipedia:Verifiability

Specialized type systems

Many type systems have been created that are specialized for use in certain environments with certain types of data, or for out-of-band static program analysis. Frequently, these are based on ideas from formal type theory and are only available as part of prototype research systems.

Dependent types

Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. For example, $matrix(3, 3)$ might be the type of a 3×3 matrix. We can then define typing rules such as the following rule for matrix multiplication:

$$matrix_{multiply} : matrix(k, m) \times matrix(m, n) \rightarrow matrix(k, n)$$

where k, m, n are arbitrary positive integer values. A variant of ML called Dependent ML has been created based on this type system, but because type checking for conventional dependent types is undecidable, not all programs using them can be type-checked without some kind of limits. Dependent ML limits the sort of equality it can decide to Presburger arithmetic.

Other languages such as Epigram make the value of all expressions in the language decidable so that type checking can be decidable. However, in general proof of decidability is undecidable, so many programs require hand-written annotations, which may be very non-trivial. As this impedes the development process many language implementations provide an easy way out in the form of an option to disable this condition. This, however, comes at the cost of making the type-checker run in an infinite loop when fed programs that don't type-check, causing the compiler to hang.

Linear types

Linear types, based on the theory of linear logic, and closely related to uniqueness types, are types assigned to values having the property that they have one and only one reference to them at all times. These are valuable for describing large immutable values such as files, strings, and so on, because any operation that simultaneously destroys a linear object and creates a similar object (such as '`str = str + "a"`') can be optimized "under the hood" into an in-place mutation. Normally this is not possible, as such mutations could cause side effects on parts of the program holding other references to the object, violating referential transparency. They are also used in the prototype operating system Singularity for interprocess communication, statically ensuring that processes cannot share objects in shared memory in order to prevent race conditions. The Clean language (a Haskell-like language) uses this type system in order to gain a lot of speedWikipedia:Verifiability while remaining safe.

Intersection types

Intersection types are types describing values that belong to *both* of two other given types with overlapping value sets. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting *either* signed or unsigned chars, because it is compatible with both types.

Intersection types are useful for describing overloaded function types: For example, if "`int` → `int` " is the type of functions taking an integer argument and returning an integer, and "`float` → `float` " is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "`int` → `int` " function safely; it simply would not use the "`float` → `float` " functionality.

In a subclassing hierarchy, the intersection of a type and an ancestor type (such as its parent) is the most derived type. The intersection of sibling types is empty.

The Forsythe language includes a general implementation of intersection types. A restricted form is refinement types.

Union types

Union types are types describing values that belong to *either* of two types. For example, in C, the signed char has range -128 to 127, and the unsigned char has range 0 to 255, so the union of these two types would have range -128 to 255. Any function handling this union type would have to deal with integers in this complete range. More generally, the only valid operations on a union type are operations that are valid on *both* types being unioned. C's "union" concept is similar to union types, but is not typesafe, as it permits operations that are valid on *either* type, rather than *both*. Union types are important in program analysis, where they are used to represent symbolic values whose exact nature (e.g., value or type) is not known.

In a subclassing hierarchy, the union of a type and an ancestor type (such as its parent) is the ancestor type. The union of sibling types is a subtype of their common ancestor (that is, all operations permitted on their common ancestor are permitted on the union type, but they may also have other valid operations in common).

Existential types

Existential types are frequently used in connection with record types to represent modules and abstract data types, due to their ability to separate implementation from interface. For example, the type "T = ∃X { a: X; f: (X → int); }" describes a module interface that has a data member named *a* of type *X* and a function named *f* that takes a parameter of the *same* type *X* and returns an integer. This could be implemented in different ways; for example:

- intT = { a: int; f: (int → int); }
- floatT = { a: float; f: (float → int); }

These types are both subtypes of the more general existential type T and correspond to concrete implementation types, so any value of one of these types is a value of type T. Given a value "t" of type "T", we know that "t.f(t.a)" is well-typed, regardless of what the abstract type *X* is. This gives flexibility for choosing types suited to a particular implementation while clients that use only values of the interface type—the existential type—are isolated from these choices.

In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example intT { a: int; f: (int → int); } could also have the type ∃X { a: X; f: (int → int); }. The simplest solution is to annotate every module with its intended type, e.g.:

- intT = { a: int; f: (int → int); } **as** ∃X { a: X; f: (X → int); }

Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type".[15] The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification.

Explicit or implicit declaration and inference

For more details on this topic, see Type inference.

Many static type systems, such as those of C and Java, require *type declarations*: The programmer must explicitly associate each variable with a particular type. Others, such as Haskell's, perform *type inference*: The compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function `f(x, y)` that adds `x` and `y` together, the compiler can infer that `x` and `y` must be numbers – since addition is only defined for numbers. Therefore, any call to `f` elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.

Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression `3.14` might imply a type of floating-point, while `[1, 2, 3]` might imply a list of integers – typically an array.

Type inference is in general possible, if it is decidable in the type theory in question. Moreover, even if inference is undecidable in general for a given type theory, inference is often possible for a large subset of real-world programs. Haskell's type system, a version of Hindley-Milner, is a restriction of System Fω to so-called rank-1 polymorphic types, in which type inference is decidable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference undecidable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.)

Types of types

Main article: Data type

A *type of types* is a kind. Kinds appear explicitly in typeful programming, such as a *type constructor* in the Haskell language.

Types fall into several broad categories:

- Primitive types – the simplest kind of type; e.g., integer and floating-point number
  - Boolean
  - Integral types – types of whole numbers; e.g., integers and natural numbers
  - Floating point types – types of numbers in floating-point representation
- Reference types
- Option types
  - Nullable types
- Composite types – types composed of basic types; e.g., arrays or records.

  Abstract data types
- Algebraic types
- Subtype
- Derived type
- Object types; e.g., type variable
- Partial type
- Recursive type
- Function types; e.g., binary functions
- universally quantified types, such as parameterized types
- existentially quantified types, such as modules
- Refinement types – types that identify subsets of other types
- Dependent types – types that depend on terms (values)
- Ownership types – types that describe or constrain the structure of object-oriented systems
- Pre-defined types provided for convenience in real-world applications, such as date, time and money.

Unified type system

Some languages like C# have a unified type system.[16] This means that all C# types including primitive types inherit from a single root object. Every type in C# inherits from the Object class. Java has several primitive types that are not objects. Java provides wrapper object types that exist together with the primitive types so developers can use either the wrapper object types or the simpler non-object primitive types.

Compatibility: equivalence and subtyping

A type-checker for a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For instance, in an assignment statement of the form `x := e`, the inferred type of the expression `e` must be consistent with the declared or inferred type of the variable `x`. This notion of consistency, called *compatibility*, is specific to each programming language.

If the type of `e` and the type of `x` are the same and assignment is allowed for that type, then this is a valid expression. In the simplest type systems, therefore, the question of whether two types are compatible reduces to that of whether they are *equal* (or *equivalent*). Different languages, however, have different criteria for when two type expressions are understood to denote the same type. These different *equational theories* of types vary widely, two extreme cases being *structural type systems*, in which any two types are equivalent that describe values with the

same structure, and *nominative type systems*, in which no two syntactically distinct type expressions denote the same type (*i.e.*, types must have the same "name" in order to be equal).

In languages with subtyping, the compatibility relation is more complex. In particular, if `A` is a subtype of `B`, then a value of type `A` can be used in a context where one of type `B` is expected, even if the reverse is not true. Like equivalence, the subtype relation is defined differently for each programming language, with many variations possible. The presence of parametric or ad hoc polymorphism in a language may also have implications for type compatibility.

Programming style

Some programmers prefer statically typed languages; others prefer dynamically typed languages. Statically typed languages alert programmers to type errors during compilation, and they may perform better at runtime. Advocates of dynamically typed languages claim they better support rapid prototyping and that type errors are only a small subset of errors in a program. Likewise, there is often no need to manually declare all types in statically typed languages with type inference; thus, the need for the programmer to explicitly specify types of variables is automatically lowered for such languages.

References

[1] http://en.wikipedia.org/w/index.php?title=Template:Type_systems&action=edit

[2] Pierce 2002, p. 1: "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

[3] Cardelli 2004, p. 1: "The fundamental purpose of a type system is to prevent the occurrence of *execution errors* during the running of a program."

[4] Pierce 2002, p. 208.

[5] Infoworld 25 April 1983 (http://books.google.co.uk/books?id=7i8EAAAAMBAJ&pg=PA66&lpg=PA66&dq=pascal+type+system+"too+strong"&source=bl&ots=PGyKS1fWUb&sig=ebFI6fk_yxwyY4b7sHSklp048Q4&hl=en&ei=lSmjTunuBo6F8gPOu43CCA&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBsQ6AEwAA#v=onepage&q=pascal type system "too strong"&f=false)

[6] [[Brian Kernighan (http://www.cs.virginia.edu/~cs655/readings/bwk-on-pascal.html)]: *Why Pascal is not my favorite language*]

[7] Pierce 2002.

[8] Visual Basic is an example of a language that is both type-safe and memory-safe.

[9] Standard ECMA-262 (http://www.ecma-international.org/publications/standards/Ecma-262.htm). Ecma-international.org. Retrieved on 2013-07-17.

[10] Strict mode - JavaScript | MDN (https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/Strict_mode). Developer.mozilla.org (2013-07-03). Retrieved on 2013-07-17.

[11] Strict Mode (JavaScript) (http://msdn.microsoft.com/en-us/library/ie/br230269(v=vs.94).aspx). Msdn.microsoft.com. Retrieved on 2013-07-17.

[12] Bracha, G.: *Pluggable Types* (http://bracha.org/pluggableTypesPosition.pdf)

[13] http://stackoverflow.com/a/13414347/975097

[14] Stefan Hanenberg. ”An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time“. OOPSLA 2010.

[15] Mitchell, John C.; Plotkin, Gordon D.; *Abstract Types Have Existential Type* (http://theory.stanford.edu/~jcm/papers/mitch-plotkin-88.pdf), ACM Transactions on Programming Languages and Systems, Vol. 10, No. 3, July 1988, pp. 470–502

[16] Standard ECMA-334 (http://www.ecma-international.org/publications/standards/Ecma-334.htm), 8.2.4 Type system unification.

Further reading

- Cardelli, Luca; Wegner, Peter (December 1985). "On Understanding Types, Data Abstraction, and Polymorphism" (http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf). *ACM Computing Surveys* (New York, NY, USA: ACM) **17** (4): 471–523. doi: 10.1145/6041.6042 (http://dx.doi.org/10.1145/6041.6042). ISSN 0360-0300 (http://www.worldcat.org/issn/0360-0300).
- Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. ISBN 978-0-262-16209-8.
- Cardelli, Luca (2004). "Type systems" (http://lucacardelli.name/Papers/TypeSystems.pdf). In Allen B. Tucker. *CRC Handbook of Computer Science and Engineering* (2nd ed.). CRC Press. ISBN 158488360X.
- Tratt, Laurence, *Dynamically Typed Languages* (http://tratt.net/laurie/research/publications/html/tratt__dynamically_typed_languages/), Advances in Computers, Vol. 77, pp. 149–184, July 2009

External links

- Smith, Chris, *What To Know Before Debating Type Systems* (http://cdsmith.wordpress.com/2011/01/09/an-old-article-i-wrote/)

Strongly typed programming language

In computer programming, programming languages are often colloquially referred to as **strongly typed** or **weakly typed**. In general, these terms do not have a precise definition. Rather, they tend to be used by advocates or critics of a given programming language, as a means of explaining why a given language is better or worse than alternatives.

History

In 1974, Liskov and Zilles described a strong-typed language as one in which "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function." Jackson wrote, "In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types."

Definitions of "strong" or "weak"

A number of different language design decisions have been referred to as evidence of "strong" or "weak" typing. In fact, many of these are more accurately understood as the presence or absence of type safety, memory safety, static type-checking, or dynamic type-checking.

Implicit type conversions and "type punning"

Some programming languages make it easy to use a value of one type as if it were a value of another type. This is sometimes described as "weak typing".

For example, Aahz Maruch writes that "*Coercion occurs when you have a statically typed language and you use the syntactic features of the language to force the usage of one type as if it were a different type (consider the common use of void* in C). Coercion is usually a symptom of weak typing. Conversion, OTOH, creates a brand-new object of the appropriate type.*" [1]

As another example, GCC describes this as *type-punning* and warns that it will *break strict aliasing*. Thiago Macieira discusses several problems that can arise when type-punning causes the compiler to make inappropriate optimizations.[2]

It is easy to focus on the syntax, but Macieira's argument is really about semantics. There are many examples of languages which allow implicit conversions, but in a type-safe manner. For example, both C++ and C# allow programs to define operators to convert a value from one type to another in a semantically meaningful way. When a C++ compiler encounters such a conversion, it treats the operation just like a function call. In contrast, converting a value to the C type "void*" is an unsafe operation which is invisible to the compiler.

Pointers

Some programming languages expose pointers as if they were numeric values, and allow users to perform arithmetic on them. These languages are sometimes referred to as "weakly typed", since pointer arithmetic can be used to bypass the language's type system.

Untagged unions

Some programming languages support untagged unions, which allow a value of one type to be viewed as if it were a value of another type. In the article titled *A hacked Boolean*, Bill McCarthy demonstrates how a Boolean value in .NET programming may become internally corrupted so that two values may both be "true" and yet still be considered unequal to each other.[3]

Dynamic type-checking

Some programming languages do not have static type-checking. In many such languages, it is easy to write programs which would be rejected by most static type-checkers. For example, a variable might store either a number or the Boolean value "false". Some programmers refer to these languages as "weakly typed", since they do not *seem* to enforce the "strong" type discipline found in a language with a static type-checker.

Static type-checking

In Luca Cardelli's article Typeful Programming,[4] a "strong type system" is described as one in which there is no possibility of an unchecked runtime type error. In other writing, the absence of unchecked run-time errors is referred to as *safety* or *type safety*; Tony Hoare's early papers call this property *security*.

Predictability

Some programmers refer to a language as "weakly typed" if simple operations do not behave in a way that they would expect. For example, consider the following program:

```
x = "5" + 6
```

Different languages will assign a different value to 'x':

- One language might convert 6 to a string, and concatenate the two arguments to produce the string "56" (e.g. JavaScript)
- Another language might convert "5" to a number, and add the two arguments to produce the number 11 (e.g. Perl, PHP)
- Yet another language might convert the string "5" to a pointer representing where the string is stored within memory, and add 6 to that value to produce a semi-random address (e.g. C)
- And yet another language might simply fail to compile this program or run the code, saying that the two operands have incompatible type (e.g. Ruby, Python, BASIC)

Languages that work like the first three examples have all been called "weakly typed" at various times, even though only one of them (the third) represents a safety violation.

Type inference

Languages with static type systems differ to the extent that users are required to manually state the types used in their program. Some languages, such as C, require that every variable be declared with a type. Other languages, such as Haskell, use the Hindley-Milner method to infer all types based on a global analysis. Other languages, such as C# and C++, lie somewhere in between; some types can be inferred based on local information, while others must be specified. Some programmers use the term weakly typed to refer to languages with type inference, often without realizing that the type information is present but implicit.

Variation across programming languages

Note that some of these definitions are contradictory, others are merely orthogonal, and still others are special cases (with additional constraints) of other, more "liberal" (less strong) definitions. Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed. For instance:

- Java, Pascal, Ada and C require all variables to have a declared type, and support the use of explicit casts of arithmetic values to other arithmetic types. Java, C#, Ada and Pascal are sometimes said to be more strongly typed than C, a claim that is probably based on the fact that C supports more kinds of implicit conversions, and C also allows pointer values to be explicitly cast while Java and Pascal do not. Java itself may be considered more strongly typed than Pascal as manners of evading the static type system in Java are controlled by the Java Virtual Machine's type system. C# is similar to Java in that respect, though it allows disabling dynamic type checking by explicitly putting code segments in an "unsafe context". Pascal's type system has been described as "too strong", because the size of an array or string is part of its type, making some programming tasks very difficult.[5][6]
- The object-oriented programming languages Smalltalk, Ruby, Python, and Self are all "strongly typed" in the sense that typing errors are prevented at runtime and they do little implicit type conversion, but these languages make no use of static type checking: the compiler does not check or enforce type constraint rules. The term duck typing is now used to describe the dynamic typing paradigm used by the languages in this group.
- The Lisp family of languages are all "strongly typed" in the sense that typing errors are prevented at runtime. Some Lisp dialects like Common Lisp or Clojure do support various forms of type declarations[7] and some compilers (CMUCL[8] and related) use these declarations together with type inference to enable various optimizations and also limited forms of compile time type checks.
- Standard ML, F#, OCaml and Haskell are statically type checked but the compiler automatically infers a precise type for all values. These languages (along with most functional languages) are considered to have stronger type systems than Java, as they permit no implicit type conversions. While OCaml's libraries allow one form of evasion (*Object magic*), this feature remains unused in most applications.
- Visual Basic is a hybrid language. In addition to variables with declared types, it is also possible to declare a variable of "Variant" data type that can store data of any type. Its implicit casts are fairly liberal where, for example, one can sum string variants and pass the result into an integer variable.
- Assembly language and Forth have been said to be *untyped*. There is no type checking; it is up to the programmer to ensure that data given to functions is of the appropriate type. Any type conversion required is explicit.

For this reason, writers who wish to write unambiguously about type systems often eschew the term "strong typing" in favor of specific expressions such as "type safety".

References

[1] Typing: Strong vs. Weak, Static vs. Dynamic (http://www.artima.com/weblogs/viewpost.jsp?thread=7590)
[2] Type-punning and strict-aliasing, Thiago Macieira (http://blog.qt.digia.com/blog/2011/06/10/type-punning-and-strict-aliasing/)
[3] A hacked Boolean (http://msmvps.com/blogs/bill/archive/2004/06/23/8730.aspx)
[4] ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-045.pdf page 3
[5] Infoworld April 25, 1983 (http://books.google.co.uk/books?id=7i8EAAAAMBAJ&pg=PA66&lpg=PA66&dq=pascal+type+system+"too+strong"&source=bl&ots=PGyKS1fWUb&sig=ebFI6fk_yxwyY4b7sHSklp048Q4&hl=en&ei=lSmjTunuBo6F8gPOu43CCA&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBsQ6AEwAA#v=onepage&q=pascal type system "too strong"&f=false)
[6] [[Brian Kernighan (http://www.cs.virginia.edu/~cs655/readings/bwk-on-pascal.html): *Why Pascal is not my favourite language*]
[7] Common Lisp HyperSpec, Types and Classes (http://www.lispworks.com/documentation/HyperSpec/Body/04_.htm)
[8] CMUCL User's Manual: The Compiler, Types in Python (http://common-lisp.net/project/cmucl/doc/cmu-user/compiler.html#toc123)

Weak typing

In computer programming, programming languages are often colloquially referred to as **strongly typed** or **weakly typed**. In general, these terms do not have a precise definition. Rather, they tend to be used by advocates or critics of a given programming language, as a means of explaining why a given language is better or worse than alternatives.

History

In 1974, Liskov and Zilles described a strong-typed language as one in which "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function." Jackson wrote, "In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types."

Definitions of "strong" or "weak"

A number of different language design decisions have been referred to as evidence of "strong" or "weak" typing. In fact, many of these are more accurately understood as the presence or absence of type safety, memory safety, static type-checking, or dynamic type-checking.

Implicit type conversions and "type punning"

Some programming languages make it easy to use a value of one type as if it were a value of another type. This is sometimes described as "weak typing".

For example, Aahz Maruch writes that "*Coercion occurs when you have a statically typed language and you use the syntactic features of the language to force the usage of one type as if it were a different type (consider the common use of void* in C). Coercion is usually a symptom of weak typing. Conversion, OTOH, creates a brand-new object of the appropriate type.*" [1]

As another example, GCC describes this as *type-punning* and warns that it will *break strict aliasing*. Thiago Macieira discusses several problems that can arise when type-punning causes the compiler to make inappropriate optimizations.[2]

It is easy to focus on the syntax, but Macieira's argument is really about semantics. There are many examples of languages which allow implicit conversions, but in a type-safe manner. For example, both C++ and C# allow programs to define operators to convert a value from one type to another in a semantically meaningful way. When a C++ compiler encounters such a conversion, it treats the operation just like a function call. In contrast, converting a value to the C type "void*" is an unsafe operation which is invisible to the compiler.

Pointers

Some programming languages expose pointers as if they were numeric values, and allow users to perform arithmetic on them. These languages are sometimes referred to as "weakly typed", since pointer arithmetic can be used to bypass the language's type system.

Untagged unions

Some programming languages support untagged unions, which allow a value of one type to be viewed as if it were a value of another type. In the article titled *A hacked Boolean*, Bill McCarthy demonstrates how a Boolean value in .NET programming may become internally corrupted so that two values may both be "true" and yet still be considered unequal to each other.[3]

Dynamic type-checking

Some programming languages do not have static type-checking. In many such languages, it is easy to write programs which would be rejected by most static type-checkers. For example, a variable might store either a number or the Boolean value "false". Some programmers refer to these languages as "weakly typed", since they do not *seem* to enforce the "strong" type discipline found in a language with a static type-checker.

Static type-checking

In Luca Cardelli's article Typeful Programming,[4] a "strong type system" is described as one in which there is no possibility of an unchecked runtime type error. In other writing, the absence of unchecked run-time errors is referred to as *safety* or *type safety*; Tony Hoare's early papers call this property *security*.

Predictability

Some programmers refer to a language as "weakly typed" if simple operations do not behave in a way that they would expect. For example, consider the following program:

```
x = "5" + 6
```

Different languages will assign a different value to 'x':

- One language might convert 6 to a string, and concatenate the two arguments to produce the string "56" (e.g. JavaScript)
- Another language might convert "5" to a number, and add the two arguments to produce the number 11 (e.g. Perl, PHP)
- Yet another language might convert the string "5" to a pointer representing where the string is stored within memory, and add 6 to that value to produce a semi-random address (e.g. C)
- And yet another language might simply fail to compile this program or run the code, saying that the two operands have incompatible type (e.g. Ruby, Python, BASIC)

Languages that work like the first three examples have all been called "weakly typed" at various times, even though only one of them (the third) represents a safety violation.

Type inference

Languages with static type systems differ to the extent that users are required to manually state the types used in their program. Some languages, such as C, require that every variable be declared with a type. Other languages, such as Haskell, use the Hindley-Milner method to infer all types based on a global analysis. Other languages, such as C# and C++, lie somewhere in between; some types can be inferred based on local information, while others must be specified. Some programmers use the term weakly typed to refer to languages with type inference, often without

realizing that the type information is present but implicit.

Variation across programming languages

Note that some of these definitions are contradictory, others are merely orthogonal, and still others are special cases (with additional constraints) of other, more "liberal" (less strong) definitions. Because of the wide divergence among these definitions, it is possible to defend claims about most programming languages that they are either strongly or weakly typed. For instance:

- Java, Pascal, Ada and C require all variables to have a declared type, and support the use of explicit casts of arithmetic values to other arithmetic types. Java, C#, Ada and Pascal are sometimes said to be more strongly typed than C, a claim that is probably based on the fact that C supports more kinds of implicit conversions, and C also allows pointer values to be explicitly cast while Java and Pascal do not. Java itself may be considered more strongly typed than Pascal as manners of evading the static type system in Java are controlled by the Java Virtual Machine's type system. C# is similar to Java in that respect, though it allows disabling dynamic type checking by explicitly putting code segments in an "unsafe context". Pascal's type system has been described as "too strong", because the size of an array or string is part of its type, making some programming tasks very difficult.[5][6]
- The object-oriented programming languages Smalltalk, Ruby, Python, and Self are all "strongly typed" in the sense that typing errors are prevented at runtime and they do little implicit type conversion, but these languages make no use of static type checking: the compiler does not check or enforce type constraint rules. The term duck typing is now used to describe the dynamic typing paradigm used by the languages in this group.
- The Lisp family of languages are all "strongly typed" in the sense that typing errors are prevented at runtime. Some Lisp dialects like Common Lisp or Clojure do support various forms of type declarations[7] and some compilers (CMUCL[8] and related) use these declarations together with type inference to enable various optimizations and also limited forms of compile time type checks.
- Standard ML, F#, OCaml and Haskell are statically type checked but the compiler automatically infers a precise type for all values. These languages (along with most functional languages) are considered to have stronger type systems than Java, as they permit no implicit type conversions. While OCaml's libraries allow one form of evasion (*Object magic*), this feature remains unused in most applications.
- Visual Basic is a hybrid language. In addition to variables with declared types, it is also possible to declare a variable of "Variant" data type that can store data of any type. Its implicit casts are fairly liberal where, for example, one can sum string variants and pass the result into an integer variable.
- Assembly language and Forth have been said to be *untyped*. There is no type checking; it is up to the programmer to ensure that data given to functions is of the appropriate type. Any type conversion required is explicit.

For this reason, writers who wish to write unambiguously about type systems often eschew the term "strong typing" in favor of specific expressions such as "type safety".

References

[1] Typing: Strong vs. Weak, Static vs. Dynamic (http://www.artima.com/weblogs/viewpost.jsp?thread=7590)
[2] Type-punning and strict-aliasing, Thiago Macieira (http://blog.qt.digia.com/blog/2011/06/10/type-punning-and-strict-aliasing/)
[3] A hacked Boolean (http://msmvps.com/blogs/bill/archive/2004/06/23/8730.aspx)
[4] ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-045.pdf page 3
[5] Infoworld April 25, 1983 (http://books.google.co.uk/books?id=7i8EAAAAMBAJ&pg=PA66&lpg=PA66&dq=pascal+type+system+"too+strong"&source=bl&ots=PGyKS1fWUb&sig=ebFI6fk_yxwyY4b7sHSklp048Q4&hl=en&ei=lSmjTunuBo6F8gPOu43CCA&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBsQ6AEwAA#v=onepage&q=pascal type system "too strong"&f=false)
[6] [[Brian Kernighan (http://www.cs.virginia.edu/~cs655/readings/bwk-on-pascal.html): *Why Pascal is not my favourite language*]
[7] Common Lisp HyperSpec, Types and Classes (http://www.lispworks.com/documentation/HyperSpec/Body/04_.htm)
[8] CMUCL User's Manual: The Compiler, Types in Python (http://common-lisp.net/project/cmucl/doc/cmu-user/compiler.html#toc123)

Syntax

In computer science, the **syntax** of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. This applies both to programming languages, where the document represents source code, and markup languages, where the document represents data. The syntax of a language defines its surface form. Text-based computer languages are based on sequences of characters, while visual programming languages are based on the spatial layout and connections between symbols (which may be textual or graphical). Documents that are syntactically invalid are said to have a syntax error.

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting and indent style are often used to aid programmers in recognizing elements of source code. Color coded highlighting is used in this piece of code written in Python.

Syntax – the form – is contrasted with semantics – the meaning. In processing computer languages, semantic processing generally comes after syntactic processing, but in some cases semantic processing is necessary for complete syntactic analysis, and these are done together or concurrently. In a compiler, the syntactic analysis comprises the frontend, while semantic analysis comprises the backend (and middle end, if this phase is distinguished).

Levels of syntax

Computer language syntax is generally distinguished into three levels:

- Words – the lexical level, determining how characters form tokens;
- Phrases – the grammar level, narrowly speaking, determining how tokens form phrases;
- Context – determining what objects or variables names refer to, if types are valid, etc.

Distinguishing in this way yields modularity, allowing each level to be described and processed separately, and often independently. First a lexer turns the linear sequence of characters into a linear sequence of tokens; this is known as "lexical analysis" or "lexing". Second the parser turns the linear sequence of tokens into a hierarchical syntax tree; this is known as "parsing" narrowly speaking. Thirdly the contextual analysis resolves names and checks types. This modularity is sometimes possible, but in many real-world languages an earlier step depends on a later step – for example, the lexer hack in C is because tokenization depends on context. Even in these cases, syntactical analysis is often seen as approximating this ideal model.

The parsing stage itself can be divided into two parts: the parse tree or "concrete syntax tree" which is determined by the grammar, but is generally far too detailed for practical use, and the abstract syntax tree (AST), which simplifies this into a usable form. The AST and contextual analysis steps can be considered a form of semantic analysis, as they are adding meaning and interpretation to the syntax, or alternatively as informal, manual implementations of syntactical rules that would be difficult or awkward to describe or implement formally.

The levels generally correspond to levels in the Chomsky hierarchy. Words are in a regular language, specified in the lexical grammar, which is a Type-3 grammar, generally given as regular expressions. Phrases are in a context-free language (CFL), generally a deterministic context-free language (DCFL), specified in a phrase structure grammar,

which is a Type-2 grammar, generally given as production rules in Backus–Naur Form (BNF). Phrase grammars are often specified in much more constrained grammars than full context-free grammars, in order to make them easier to parse; while the LR parser can parse any DCFL in linear time, the simple LALR parser and even simpler LL parser are more efficient, but can only parse grammars whose production rules are constrained. Contextual structure can in principle be described by a context-sensitive grammar, and automatically analyzed by means such as attribute grammars, though in general this step is done manually, via name resolution rules and type checking, and implemented via a symbol table which stores names and types for each scope.

Tools have been written that automatically generate a lexer from a lexical specification written in regular expressions and a parser from the phrase grammar written in BNF: this allows one to use declarative programming, rather than need to have procedural or functional programming. A notable example is the lex-yacc pair. These automatically produce a *concrete* syntax tree; the parser writer must then manually write code describing how this is converted to an *abstract* syntax tree. Contextual analysis is also generally implemented manually. Despite the existence of these automatic tools, parsing is often implemented manually, for various reasons – perhaps the phrase structure is not context-free, or an alternative implementation improves performance or error-reporting, or allows the grammar to be changed more easily. Parsers are often written in functional languages, such as Haskell, in scripting languages, such as Python or Perl, or in C or C++.

Examples of errors

Main article: Syntax error

As an example, `(add 1 1)` is a syntactically valid Lisp program (assuming the 'add' function exists, else name resolution fails), adding 1 and 1. However, the following are invalid:

```
(_ 1 1)    lexical error: '_' is not valid
(add 1 1   parsing error: missing closing ')'
(add 1 x)  name error: 'x' is not bound
```

Note that the lexer is unable to identify the error – all it knows is that, after producing the token LEFT_PAREN, '(' the remainder of the program is invalid, since no word rule begins with '_'. At the parsing stage, the parser has identified the "list" production rule due to the '(' token (as the only match), and thus can give an error message; in general it may be ambiguous. At the context stage, the symbol 'x' exists in the syntax tree, but has not been defined, and thus the context analyzer can give a specific error.

In a strongly typed language, type errors are also a form of syntax error which is generally determined at the contextual analysis stage, and this is considered a strength of strong typing. For example, the following is syntactically invalid Python code (as these are literals, the type can be determined at parse time):

```
'a' + 1
```

…as it adds a string and an integer. This can be detected at the parsing (phrase analysis) level if one has separate rules for "string + string" and "integer + integer", but more commonly this will instead be parsed by a general rule like "LiteralOrIdentifier + LiteralOrIdentifier" and then the error will be detected at contextual analysis stage, where type checking occurs. In some cases this validation is not done, and these syntax errors are only detected at runtime.

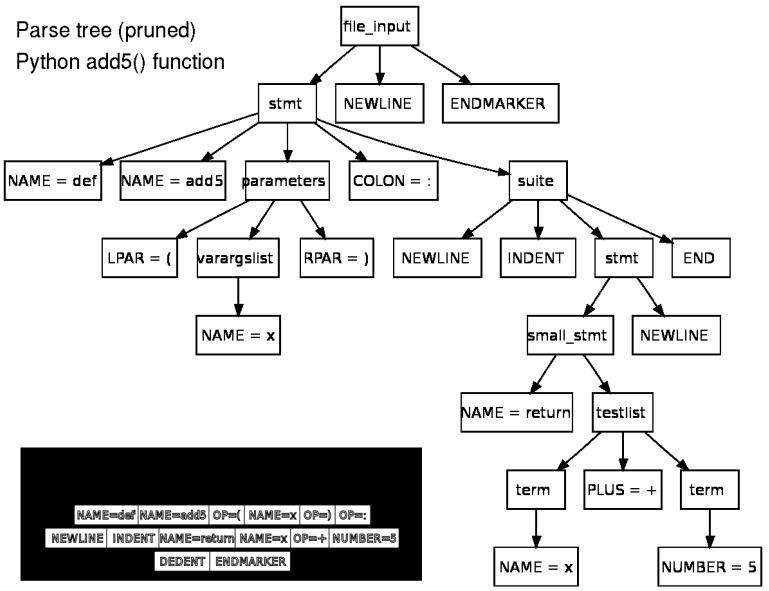
In a weakly typed language, where type can only be determined at runtime, type errors are instead a semantic error, and can only be determined at runtime. The following Python code:

```
a + b
```

is ambiguous, and while syntactically valid at the phrase level, it can only be validated at runtime, as variables do not have type in Python, only values do.

Syntax definition

The syntax of textual programming languages is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure) to inductively specify syntactic categories (nonterminals) and *terminal* symbols. Syntactic categories are defined by rules called *productions*, which specify the values that belong to a particular syntactic category. Terminal symbols are the concrete characters or strings of characters (for example keywords such as *define*, *if*, *let*, or *void*) from which syntactically valid programs are constructed.



Parse tree of Python code with inset tokenization

A language can have different equivalent grammars, such as equivalent regular expressions (at the lexical levels), or different phrase rules which generate the same language. Using a broader category of grammars, such as LR grammars, can allow shorter or simpler grammars compared with more restricted categories, such as LL grammar, which may requires longer grammars with more rules. Different but equivalent phrase grammars yield different parse trees, though the underlying language (set of valid documents) is the same.

Example: Lisp

Below is a simple grammar, defined using the notation of regular expressions and Backus–Naur Form. It describes the syntax of Lisp, which defines productions for the syntactic categories *expression*, *atom*, *number*, *symbol*, and *list*:

```
expression ::= atom   | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
list       ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

Here the decimal digits, upper- and lower-case characters, and parentheses are terminal symbols.

The following are examples of well-formed token sequences in this grammar: `'12345'`,`'()'`,`'(a b c232 (1))'`

Complex grammars

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The phrase grammar of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars,[1] though the overall syntax is context-sensitive (due to variable declarations and nested scopes), hence Type-1. However, there are exceptions, and for some languages the phrase grammar is Type-0 (Turing-complete).

In some languages like Perl and Lisp the specification (or implementation) of the language allows constructs that execute during the parsing phase. Furthermore, these languages have constructs that allow the programmer to alter the behavior of the parser. This combination effectively blurs the distinction between parsing and execution, and makes syntax analysis an undecidable problem in these languages, meaning that the parsing phase may not finish. For example, in Perl it is possible to execute code during parsing using a `BEGIN` statement, and Perl function prototypes may alter the syntactic interpretation, and possibly even the syntactic validity of the remaining code.[2] Colloquially this is referred to as "only Perl can parse Perl" (because code must be executed during parsing, and can modify the grammar), or more strongly "even Perl cannot parse Perl" (because it is undecidable). Similarly, Lisp macros introduced by the `defmacro` syntax also execute during parsing, meaning that a Lisp compiler must have an entire Lisp run-time system present. In contrast C macros are merely string replacements, and do not require code execution.

Syntax versus semantics

The syntax of a language describes the form of a valid program, but does not provide any information about the meaning of the program or the results of executing that program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because `p` is a null pointer, the operations `p->real` and `p->im` have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

More simply:

```
int x;
printf("%d", x);
```

is syntactically valid, but not semantically defined, as it uses an uninitialized variable.

References

[1] Section 2.2: Pushdown Automata, pp.101–114.

[2] The following discussions give examples:

- Perl and Undecidability (http://www.jeffreykegler.com/Home/perl-and-undecidability)
- LtU comment clarifying that the undecidable problem is membership in the class of Perl programs (http://lambda-the-ultimate.org/node/3564#comment-50578)
- chromatic's example of Perl code that gives a syntax error depending on the value of random variable (http://www.modernperlbooks.com/mt/2009/08/on-parsing-perl-5.html)

External links

- Various syntactic constructs used in computer programming languages (http://merd.sourceforge.net/pixel/language-study/syntax-across-languages/)

Scripting language

A **scripting language** or **script language** is a programming language that supports **scripts**, programs written for a special run-time environment that can interpret (rather than compile) and automate the execution of tasks that could alternatively be executed one-by-one by a human operator. Environments that can be automated through scripting include software applications, web pages within a web browser, the shells of operating systems (OS), and embedded systems. A scripting language can be viewed as a domain-specific language for a particular environment; in the case of scripting an application, this is also known as an **extension language**. Scripting languages are also sometimes referred to as very high-level programming languages, as they operate at a high level of abstraction, or as **control languages**, particularly for job control languages on mainframes.

The term "scripting language" is also used loosely to refer to dynamic high-level general-purpose language, such as Perl, Tcl, and Python,[1] with the term "script" often used for small programs (up to a few thousand lines of code) in such languages, or in domain-specific languages such as the text-processing languages sed and AWK. Some of these languages were originally developed for use within a particular environment, and later developed into portable domain-specific or general-purpose languages. Conversely, many general-purpose languages have dialects that are used as scripting languages. This article discusses scripting languages in the narrow sense of languages for a specific environment; dynamic, general-purpose, and high-level languages are discussed at those articles.

The spectrum of scripting languages ranges from very small and highly domain-specific languages to general-purpose programming languages used for scripting. Standard examples of scripting languages for specific environments include: Bash, for the Unix or Unix-like operating systems; ECMAScript (JavaScript), for web browsers; and Visual Basic for Applications, for Microsoft Office applications. Lua is a language designed and widely used as an extension language. Python is a general-purpose language that is also commonly used as an extension language, while ECMAScript is still primarily a scripting language for web browsers, but is also used as a general-purpose language. The Emacs Lisp dialect of Lisp (for the Emacs editor) and the Visual Basic for Applications dialect of Visual Basic are examples of scripting language dialects of general-purpose languages. Some game systems, notably the Trainz franchise of Railroad simulators have been extensively extended in functionality by scripting extensions.

Characteristics

In principle any language can be used as a scripting language, given libraries or bindings for a specific environment. Formally speaking, "scripting" is a property of the primary implementations and uses of a language, hence the ambiguity about whether a language "is" a scripting language for languages with multiple implementations. However, many languages are not very suited for use as scripting languages and are rarely if ever used as such.

Typically scripting languages are intended to be very fast to pick up and author programs in. This generally implies relatively simple syntax and semantics. For example, it is uncommon to use Java as a scripting language due to the lengthy syntax and restrictive rules about which classes exist in which files – contrast to Python, where it is possible to briefly define some functions in a file. A scripting language is usually interpreted from source code or bytecode. By contrast, the software environment the scripts are written for is typically written in a compiled language and distributed in machine code form. Scripting languages may be designed for use by end users of a program – end-user development – or may be only for internal use by developers, so they can write portions of the program in the scripting language. Scripting languages abstract their users from variable types and memory management.

Scripts are often created or modified by the person executing them,[2] though they are also often distributed, such as when large portions of games are written in a scripting language. In many implementations a script or portions of one may be executed interactively on a command line.

History

Early mainframe computers (in the 1950s) were non-interactive, instead using batch processing. IBM's Job Control Language (JCL) is the archetype of languages used to control batch processing.

The first interactive shells were developed in the 1960s to enable remote operation of the first time-sharing systems, and these used shell scripts, which controlled running computer programs within a computer program, the shell. Calvin Mooers in his TRAC language is generally credited with inventing *command substitution*, the ability to embed commands in scripts that when interpreted insert a character string into the script. Multics calls these *active functions*. Louis Pouzin wrote an early processor for command scripts called RUNCOM for CTSS around 1964. Stuart Madnick at MIT wrote a scripting language for IBM's CP/CMS in 1966. He originally called this processor COMMAND, later named EXEC. Multics included an offshoot of CTSS RUNCOM, also called RUNCOM. EXEC was eventually replaced by EXEC 2 and REXX.

Languages such as Tcl and Lua were specifically designed as general purpose scripting languages that could be embedded in any application. Other languages such as Visual Basic for Applications (VBA) provided strong integration with the automation facilities of an underlying system. Embedding of such general purpose scripting languages instead of developing a new language for each application also had obvious benefits, relieving the application developer of the need to code a language translator from scratch and allowing the user to apply skills learned elsewhere.

Some software incorporates several different scripting languages. Modern web browsers typically provide a language for writing extensions to the browser itself, and several standard embedded languages for controlling the browser, including JavaScript (a dialect of ECMAScript) or XUL.

Types of scripting languages

Glue languages

Scripting is often contrasted with system programming, as in Ousterhout's dichotomy or "programming in the large and programming in the small". In this view, scripting is particularly glue code, connecting system components, and a language specialized for this purpose is a glue language. Pipelines and shell scripting are archetypal examples of glue languages, and Perl was initially developed to fill this same role. Web development can be considered a use of

glue languages, interfacing between a database and web server. The characterization of glue languages as scripting languages is ambiguous, however, as if a substantial amount of logic is part of the "glue" code, it is better characterized as simply another software component.

A **glue language** is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining:

- Custom commands for a command shell
- Smaller programmes than those that are better implemented in a compiled language
- "Wrapper" programmes for executables, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data base, assembler, compiler, etc.
- Scripts that may change
- Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

Glue language examples:

- Erlang
- Unix Shell scripts (ksh, csh, bash, sh and others)
- Windows PowerShell
- ecl
- DCL
- Scheme
- JCL
- m4
- VBScript
- JScript and JavaScript
- AppleScript
- Python
- Ruby
- Lua
- Tcl
- Perl
- PHP
- Pure
- REXX
- XSLT

Macro languages exposed to operating system or application components can serve as glue languages. These include Visual Basic for Applications, WordBasic, LotusScript, CorelScript, PerfectScript, Hummingbird Basic, QuickScript, SaxBasic, and WinWrap Basic. Other tools like awk can also be considered glue languages, as can any language implemented by an ActiveX WSH engine (VBScript, JScript and VBA by default in Windows and third-party engines including implementations of Rexx, Perl, Tcl, Python, XSLT, Ruby, Delphi, &c). A majority of applications can access and use operating system components via the object models or its own functions.

Other devices like programmable calculators may also have glue languages; the operating systems of PDAs such as Windows CE may have available native or third-party macro tools that glue applications together, in addition to implementations of common glue languages—including Windows NT, MS-DOS and some Unix shells, Rexx, PHP, and Perl. Depending upon the OS version, WSH and the default script engines (VBScript and JScript) are available.

Programmable calculators can be programmed in glue languages in three ways. For example, the Texas Instruments TI-92, by factory default can be programmed with a command script language. Inclusion of the scripting and glue language Lua in the TI-NSpire series of calculators could be seen as a successor to this. The primary on-board

high-level programming languages of most graphing calculators (most often Basic variants, sometimes Lisp derivatives, and more uncommonly, C derivatives) in many cases can glue together calculator functions—such as graphs, lists, matrices, etc. Third-party implementations of more comprehensive Basic version that may be closer to variants listed as glue languages in this article are available—and attempts to implement Perl, Rexx, or various operating system shells on the TI and HP graphing calculators are also mentioned. PC-based C cross-compilers for some of the TI and HP machines used in conjunction with tools that convert between C and Perl, Rexx, awk, as well as shell scripts to Perl, VBScript to and from Perl make it possible to write a programme in a glue language for eventual implementation (as a compiled programme) on the calculator.

Job control languages and shells

Main article: Shell script

A major class of scripting languages has grown out of the automation of job control, which relates to starting and controlling the behavior of system programs. (In this sense, one might think of shells as being descendants of IBM's JCL, or Job Control Language, which was used for exactly this purpose.) Many of these languages' interpreters double as command-line interpreters such as the Unix shell or the MS-DOS `COMMAND.COM`. Others, such as AppleScript offer the use of English-like commands to build scripts.

GUI scripting

With the advent of graphical user interfaces, a specialized kind of scripting language emerged for controlling a computer. These languages interact with the same graphic windows, menus, buttons, and so on that a human user would. They do this by simulating the actions of a user. These languages are typically used to automate user actions. Such languages are also called "macros" when control is through simulated key presses or mouse clicks.

These languages could in principle be used to control any GUI application; but, in practice their use is limited because their use needs support from the application and from the operating system. There are a few exceptions to this limitation. Some GUI scripting languages are based on recognizing graphical objects from their display screen pixels. These GUI scripting languages do not depend on support from the operating system or application.

Application-specific languages

Many large application programs include an idiomatic scripting language tailored to the needs of the application user. Likewise, many computer game systems use a custom scripting language to express the programmed actions of non-player characters and the game environment. Languages of this sort are designed for a single application; and, while they may superficially resemble a specific general-purpose language (e.g. QuakeC, modeled after C), they have custom features that distinguish them. Emacs Lisp, while a fully formed and capable dialect of Lisp, contains many special features that make it most useful for extending the editing functions of Emacs. An application-specific scripting language can be viewed as a domain-specific programming language specialized to a single application.

Extension/embeddable languages

A number of languages have been designed for the purpose of replacing application-specific scripting languages by being embeddable in application programs. The application programmer (working in C or another systems language) includes "hooks" where the scripting language can control the application. These languages may be technically equivalent to an application-specific extension language but when an application embeds a "common" language, the user gets the advantage of being able to transfer skills from application to application. A more generic alternative is simply to provide a library (often a C library) that a general-purpose language can use to control the application, without modifying the language for the specific domain.

JavaScript began as and primarily still is a language for scripting inside web browsers; however, the standardization of the language as ECMAScript has made it popular as a general purpose embeddable language. In particular, the

Mozilla implementation SpiderMonkey is embedded in several environments such as the Yahoo! Widget Engine. Other applications embedding ECMAScript implementations include the Adobe products Adobe Flash (ActionScript) and Adobe Acrobat (for scripting PDF files).

Tcl was created as an extension language but has come to be used more frequently as a general purpose language in roles similar to Python, Perl, and Ruby. On the other hand, Rexx was originally created as a job control language, but is widely used as an extension language as well as a general purpose language. Perl is a general-purpose language, but had the Oraperl (1990) dialect, consisting of a Perl 4 binary with Oracle Call Interface compiled in. This has however since been replaced by a library (Perl Module), DBD::Oracle [3] [4][5].

Other complex and task-oriented applications may incorporate and expose an embedded programming language to allow their users more control and give them more functionality than can be available through a user interface, no matter how sophisticated. For example, Autodesk Maya 3D authoring tools embed the MEL scripting language, or Blender which uses Python to fill this role.

Some other types of applications that need faster feature addition or tweak-and-run cycles (e.g. game engines) also use an embedded language. During the development, this allows them to prototype features faster and tweak more freely, without the need for the user to have intimate knowledge of the inner workings of the application or to rebuild it after each tweak (which can take a significant amount of time). The scripting languages used for this purpose range from the more common and more famous Lua and Python to lesser-known ones such as AngelScript and Squirrel.

Ch is another C compatible scripting option for the industry to embed into C/C++ application programs.

References

[1] Programming is Hard, Let's Go Scripting... (http://www.perl.com/pub/2007/12/06/soto-11.html), Larry Wall, December 6, 2007

[2] IEEE Computer, 2008, *In praise of scripting* (http://www.cse.wustl.edu/~loui/praiseieee.html), Ronald Loui author

[3] https://metacpan.org/module/DBD::Oracle

[4] Oraperl (https://metacpan.org/module/Oraperl), CPAN]

[5] Perl (http://www.orafaq.com/wiki/Perl), *Underground Oracle FAQ*

External links

- Patterns for Scripted Applications (https://web.archive.org/web/20041010125419/www.doc.ic.ac.uk/~np2/patterns/scripting/) at the Wayback Machine (archived October 10, 2004)

Article Sources and Contributors

Computer programming *Source*: http://en.wikipedia.org/w/index.php?oldid=608596926 *Contributors*: *drew, 1exec1, 206.26.152.xxx, 209.157.137.xxx, 64.24.16.xxx, 84user, 9258fahsflkh917fas, A.amitkumar, ABF, AGK, AKGhetto, AbstractClass, Acalamari, Acdx, Acroterion, AdamCox9, Adrignola, Adw2000, Aeram16, Aeternus, AgentCDE, Ahmlmh, Ahoerstemeier, Aitias, Akanemoto, Al Lemos, Alan Liefting, AlanH, Alansohn, Alberto Orlandini, Alex.chris111, Alex.g, AlistairMcMillan, AllCalledByGod, Alyssa3467, Amiodarone, Amire80, Anbu121, Ancheta Wis, Andonic, Andrejj, Andres, AndrewHowse, Andrewman327, Andrewpmk, Andy Dingley, Angrysockhop, AnnaFrance, Antonielly, Antonio Lopez, AquaFox, Arnabatcal, ArnoldReinhold, Arvindn, Asijut, AtticusX, Auroranorth, Avoided, Awzrenn1, BD2412, Bakabaka, Banaticus, Bangsanegara, Bazonka, Beanztr, Betterworld, Bevo, BiT, Bigk105, Billiam1185, Blackworld2005, Bluemoose, Bobo192, Bonadea, Bookofjude, Bootedcat, Boson, Bougainville, Breadbaker444, Brianga, Brichard12, Brighterorange, Brother Dysk, Bubba hotep, Bucephalus, BurntSky, Butterflylunch, C.Fred, C550456, CRGreathouse, Caltas, Can't sleep, clown will eat me, CanadianLinuxUser, Cander0000, Caninja, Capi, Capi crimm, Capitalismojo, Capricorn42, Captain Disdain, Centrx, Cflm001, Cgmusselman, CharlesC, CharlotteWebb, Chazcag, Chirp Cricket, Chocolateboy, Chovain, ChrisGualtieri, ChrisLoosley, Christopher Agnew, Christopher Fairman, Chriswiki, Chuck369, Ciaccona, Clarkcj12, Closedmouth, Cmtam922, Cnilep, Cnkids, Colonies Chris, Cometstyles, Compsim, Connorthejones, Conversion script, Correogsk, Crazytales, Cstlarry, Curps, Curvers, Cybercobra, Cypherquest, CzarB, DARTH SIDIOUS 2, DMacks, DVD R W, Da phenom, Daekharel, Damian Yerrick, Damieng, DanP, Danakil, Dante Alighieri, Darkwind, Dasari12, Daverose 33, Davey-boy-10042969, DavidCary, DavidLevinson, Davidwil, Dawnseeker2000, Daydreamer302000, Dbfirs, Dcljr, Deepakr, Dekisugi, DerHexer, Derek farn, Deryck Chan, Dfmclean, Dialectric, Diberri, Diego Moya, Digitize, Discospinster, Dkasak, Doc9871, Dominic7848, Don4of4, Donald Albury, Donhalcon, Donner60, DoorsAjar, DoorsRecruiting.com, DotHectate, Dougofborg, Downtownee, Dravir5, Dreth, Drmies, Drphilharmonic, Dureo, Dusto4, Dylan Lake, Dysprosia, EAderhold, ERcheck, Ed Poor, Edderso, Editor12345678901, Edward, Edward Z. Yang, Eeekster, Eiwot, El C, ElAmericano, Elektrik Shoos, Elendal, Elf, Elkman, Emca26, Emorr280, Emperorbma, Epbr123, Ephidel, Epolk, Ericbeg, Essexmutant, Excirial, Extremist, F41t3r, Fakhr245, Falcon Kirtaran, Falcon8765, Fazdabest, Fazilati, Femto, Fg, Fgnievinski, Fraggle81, Fratrep, Frecklefoot, FreplySpang, Frosted14, FunPika, Furrykef, Fvw, GSI Webpage, Galoubet, Gamernotnerd, Gandalf61, Garik, Gary2863, Garzo, GeorgeAhad, GeorgeBills, Geremy659, Ghewgill, Ghyll, Giftlite, Gildos, Gilliam, Glacialfox, Glass Tomato, Gogo Dodo, GoodDamon, Goodvac, Goooogler, Gploc, Gracenotes, GraemeL, Graham87, Granpire Viking Man, GrantWishes, Gregsometimes, Grison, Grouf, Grouphardev, Guaka, Guanaco, Gwernol, Gökhan, Habbo sg, Hadal, Hairy Dude, Hanberke, Handbuddy, Hannes Hirzel, Happyisenough, Harvester, Headbomb, Helpmehelpmehelpme, Heltzgersworld, Helworld, Henry Flower, Hermione1980, Hernan mvs, Heron, Heymid, Hipocrite, Hustlestar4, Iamjakejakeisme, Igoldste, Ikanreed, Ikseevon, Iliketubesox, Imroy, Inanygivenhole, Intgr, InvertRect, Inzy, IronGargoyle, Isaac Oyelowo, Iuliatoyo, Ivan Štambuk, Ixfd64, J. M., J.delanoy, J3ff, JLaTondre, Jack Greenmaven, Jackal2323, Jackol, Jacob.jose, Jagged 85, JamesMoose, Jan1nad, Jarble, Jason4789, Jason5ayers, Jatos, Jaysweet, Jedediah Smith, Jeff02, Jeffrd10, Jeremylawless, Jim1138, Jmundo, Joedaddy09, Joel B. Lewis, Johnpfmcguire, Jojalozzo, Jojhutton, Josh1billion, JoshCarter15, Joshua.thomas.bird, Josve05a, Jpbowen, Jph, Jschnur, Jwestbrook, Jwh335, K.Nevelsteen, KHaskell, KJS77, KMurphyP, Kaare, Keilana, Kenny sh, Kglavin, Kifcaliph, Kkartik100, Klutzy, Kmg90, Kmurray24, Konstable, Laurusnobilis, Lee J Haywood, Leedeth, Leibniz, LeinadSpoon, Lemlwik, Lerdsuwa, Levineps, Lgrover, Liamlearns, Lieutenant of Melkor, Lightmouse, LilHelpa, LittleOldMe, LittleOldMe old, Loadmaster, Logan, Logical Gentleman, Loren.wilton, Lorenz JJ, Luckdao, Luk, Lumos3, Luna Santin, Lysander89, M4573R5H4K3, M4gnum0n, MER-C, MacMed, Macrakis, Macy, MadScientistX11, Magioladitis, Mahanga, Majilis, Marek69, Mark Renier, Markoschuetz, Maroux, Marquez, Martarius, Maryreid, Materialscientist, Matthew, Matthew Stannard, Mauler90, Maury Markowitz, Maximaximax, Mdd, Mean as custard, Melody Lavender, Mentifisto, Mesimoney, Metalhead816, Mets501, MiCovran, Michael Drüing, Michael93555, MichaelBillington, Michal Jurosz, MikeDogma, Mindmatrix, Minghong, Minimac, Minna Sora no Shita, Mipadi, Mjchonoles, Moder1235, Moosehadley, MountainRail, Mr Stephen, MrFish, MrOllie, Msikma, MusikAnimal, Mustafasr, Mwanner, Mxn, NERIUM, NHRHS2010, Nagy, Nanshu, Narayanese, Nbak, Neilc, Nephtes, Nertzy, Netkinetic, Netsnipe, Neurovelho, NewEnglandYankee, Newyorkadam, NickW557, Nigholith, Nikai, Nilesh G.Jadhav, Ningauble, Nk, Northamerica1000, Notheruser, Nothingofwater, Nubiatech, Nuno Tavares, Nwbeeson, O.Koslowski, OSXfan, Obli, Ohnoitsjamie, Optim, Optimist on the run, Orangutan, Orlady, Oxymoron83, P.jansson, PLCF, Paperfork, Paul D. Buck, Paxse, Pcu123456789, Peberdah, Pedro, Pharaoh of the Wizards, Philip Trueman, PhilipO, PhnomPencil, Phoe6, Piet Delport, Pilotguy, Pimpedshortie99, Pine, Pinethicket, Pkalkul, Plugwash, Pm2214, Poco a poco, Pointillist, Poor Yorick, Porterjoh, Poweroid, Prashanthellina, Pratyya Ghosh, Premvvnc, Prgrmr@wrk, PrimeHunter, PrisonerOfIce, Promooex, Pt9 9, Qasimb, Qirex, Qrex123, Quadell, RA0808, RCX, Ragib, Rajnishbhatt, Rama's Arrow, Ranak01, Rasmus Faber, RaveTheTadpole, Rawling, RedWolf, RedWordSmith, RexNL, Ricky15233, Rl, Rmayo, Robert Bond, Robert L, Robert Merkel, Robertinventor, Robin klein, Rodolfo miranda, Ronald mathw, Ronz, Rrelf, Rror, Rsg, Rsgeorge172, Rsrikanth05, Ruud Koot, Rwalker, Rwwww, S.K., S0ulfire84, SDC, Salv236, Sammylor095, Sanbeg, Sardanaphalus, Sasajid, Satansrubberduck, SchfiftyThree, SchreyP, Schrödinger's Neurotoxin, Schwarzbichler, SciAndTech, Scoop Dogy, Scwlong, Sean7341, Senator Palpatine, Seyda, Sgr927, Shahidsidd, Shanak, ShaunBebbers, Shirik, Silvrous, Simeon, SimonEast, Skizzik, Skr15081997, SkyWalker, Smalljim, Someone42, SpacemanSpiff, Splang, Starionwolf, Steel1943, SteinbDJ, SteloKim, Stephen Gilbert, Stephenb, Stephenbooth uk, Steven Zhang, Studerby, Suisui, SunCountryGuy01, Surfer43, Suruena, Svemir, Tablizer, TakuyaMurata, Tanalam, Tangent747, TarkusAB, Taylor Bohl, Techman224, Tedickey, Tedlin01, Teknopup, Teo64x, TestPilot, Texture, Tgeairn, The Divine Fluffalizer, The Herald, The Mighty Clod, The Thing That Should Not Be, The Transhumanist, The Transhumanist (AWB), TheIronWill, TheRanger, TheTito, Thebestofall007, Thedjatclubrock, Thedp, Thingg, Thisma, Tide rolls, Tifego, Timhowardriley, Tnxman307, Tobby72, Tobias Bergemann, Tom2we, TomasBat, TonyClarke, Touch Of Light, Tpbradbury, Trusilver, Turnstep, Tushar.kute, TuukkaH, Tweak232, Twest2665, Twsx, Tysto, Tyw7, Udayabdurrahman, Ukexpat, Umofomia, Uncle Dick, Unforgettableid, Uniquely Fabricated, Unknown Interval, Urville86, Userabc, Vanished User 8a9b4725f8376, Vanished user 9i39j3, Vasywriter, Versus22, Vipinhari, Viriditas, Vishsangale, Vladimer663, WBP Boy, Wadamja, WadeSimMiser, Wangi, Werieth, Whazzups, Whiskey in the Jar, Wickorama, Wiki alf, WikiMichel, Wikiklrsc, Wikipelli, Wilku997, Wing (usurped), Witchwooder, Wizard191, Worldeng, Wre2wre, Wtf242, Wtmitchell, Wwagner, Xagyg creator programmer, Xavier Combelle, Yellobelly64, YellowMonkey, Yintan, Yk Yk Yk, Youssefsan, Yoyosolodolocreolo, Yummifruitbat, Zeboober, Zkac050, Zoul1380, Zsinj, Zvn, Σ, 1363 anonymous edits

History of programming languages *Source*: http://en.wikipedia.org/w/index.php?oldid=608616865 *Contributors*: 1exec1, Abcarter, Acaciz, Alexanderaltman, Aljullu, Altenmann, Ancheta Wis, Andrzejsliwa, Antic-Hay, Arch dude, Arohanui, Ashawley, B3tamike, Banus, Beland, Ben Standeven, Bgeron, Capricorn42, Cassowary, CharlesGillingham, Chris the speller, ChrisGualtieri, Christophe.billiottet, Clarince63, Cleme263, Cnilep, DEddy, Danakil, Denbosch, DiarmuidPigott, Diego Moya, Discospinster, Djsasso, Duncandewar, Dylan Lake, Entropy, Epicgenius, Excirial, Firsfron, Funandtrvl, Fuzheado, Gerweck, Gf uip, Ghettoblaster, GraemeL, Greenrd, Griffin700, Hairy Dude, Happysailor, Hossein bardareh, Hotcrocodile, Huffers, ISTB351, IanOsgood, Icey, Id1337x, Indeterminate, JLaTondre, JMK, Jack Greenmaven, Jasssonpet, Jim1138, John Vandenberg, Jost Riedel, Jpbowen, Jrgdelrisco, Jrm2007, Juhovuori, KevM, Khazar2, Kooku10, KoshVorlon, Kpengboy, Lakinekaki, Lambiam, Logan, Lulu of the Lotus-Eaters, Macaldo, Mahanga, Malcolmxl5, Marc Mongenet, Mdd, Michael Fourman, Moe Epsilon, Mojo Hand, Mortense, Mr700, Nibios, Noodleki, NumberByColors, Nvourvachis, Oxymoron83, Pankaj.nith, Paulo torrens, Pgan002, Pgr94, PhilKnight, PhnomPencil, Pi, Pwjb, Quinntaylor, R. S. Shaw, Raees Iqbal, RedWolf, Reeve, Rheostatik, Rp, Rursus, Ruud Koot, Rwwww, Sabre23t, Sammi84, Sceptre, Schmettow, Sietse Snel, Simmonsandrew75, Sinbadbuddha, Skizzik, Skolastika, Skäpperöd, Sligocki, Soxey6, Steve2011, Szyslak, T-bonham, Tablizer, The Thing That Should Not Be, Tommy2010, Tony Sidaway, Torc2, Totlmstr, UdayanBanerjee, Ugog Nizdast, Ukexpat, Valodzka, Vb4ever, Wikid77, Wikiklrsc, Wikipelli, Yuriz, Zawersh, Zoicon5, 284 anonymous edits

Comparison of programming languages *Source*: http://en.wikipedia.org/w/index.php?oldid=607673620 *Contributors*: AdultSwim, Alik Kirillovich, Andreas Kaufmann, Bazonka, Bender235, Bh3u4m, BiT, Btx40, Bwoebi, Caesura, Cedar101, Christopher Kraus, Cjullien, Classicalecon, Coren, Crashie, Cybercobra, Damian Yerrick, DanBishop, EdwardH, Erik Garrison, Ethically Yours, Frietjes, Garyzx, Ghettoblaster, Giftlite, Hans Bauer, IanOsgood, J.delanoy, Jakarr, Jncraton, Jonathan de Boyne Pollard, Jwmwalrus, Kaly J., Kubanczyk, LilHelpa, Logicalvue, LuoShengli, Magioladitis, Mandarax, Maxim, Michaeldadmum, Mike92591, MizardX, Mmernex, Mr1278, Mrhmouse, Murray Langton, Neuralwarp, NevilleDNZ, Niout, OlavN, Oli Filth, Pavel Senatorov, Philipolson, RCX, Regenspaziergang, Robert Einhorn, Ronaldo Aves, Rwessel, Schneidr, Sekelsenmat, Short Circuit, Slippyd, Spoon!, Sun Creator, The Thing That Should Not Be, Tony Sidaway, Torc2, Txt.file, Ty8inf, Vadmium, Wiki Tesis, Woohookitty, Xqsd, Ysangkok, Zhaohan, Zzo38, 顏猥翁, 은빛냇물, 633 anonymous edits

Computer program *Source*: http://en.wikipedia.org/w/index.php?oldid=606564295 *Contributors*: 10metreh, 16@r, ABF, AKGhetto, AVRS, Abdullais4u, Adrianwn, AdultSwim, Ahoerstemeier, Airada, Alansohn, Aldaron, Aldie, Ale jrb, AlefZet, Aleksd, Alisha0512, AlistairMcMillan, Allan McInnes, Anaxial, Ancheta Wis, Andre Engels, Andrejj, Andres, Angrytoast, Animum, Anja98, Ans-mo, Antandrus, Arthena, Artichoker, Ash211, Atlant, Auric, Barkingdoc, Barts1a, Bcastell, Behnam8419, Bfinn, Bhadani, Blainster, Bob1960evens, Boothy443, Born2cycle, Bornhj, Bryan Derksen, Callanecc, Can't sleep, clown will eat me, CanadianLinuxUser, Cap'n Refsmmat, CardinalDan, Carrot Lord, Cartiman, Children.of.the.Kron, Chriswiki, Chun-hian, CommonsDelinker, Conversion script, CopperMurdoch, CurranH, Cybercobra, DARTH SIDIOUS 2, DMacks, DVdm, Danakil, Dannyruthe, David Kernow, Deliv3ranc3, Derek farn, DexDor, Dicklyon, Diego Moya, Diptytiw, Discospinster, DonToto, Donner60, Duncharris, ERobson, ESkog, Edward, Ehheh, ElKevbo, Elassint, EncMstr, Enchanter, Epbr123, FF2010, Fabartus, Filemon, Firemaker117, FleetCommand, Flyer22, Frencheigh, Frietjes, Frosted14, Funandtrvl, Furrykef, Gaga654, Gaius Cornelius, Ghyll, Giftlite, Greenrd, Grstain, Grunt, Guppy, Gurchzilla, Guy Harris, Guyjohnston, HappyDog, Hoo man, HotQuantum3000, Hu12, I dream of horses, I'll suck anything, Inaaaa, Incnis Mrsi, Ipsign, IslandHopper973, Islescape, Iulianu, JGNPILOT, Jaanus.kalde, JackLumber, Jackmatty, Jackol, JamesMoose, Jerome Kelly, Jerryobject, Jesant13, John Fader, John5678777, JohnWittle, JohnnyRush10, Jonnyapple, Josh Parris, Josve05a, Jotomicron, Jpbowen, Jpgordon, Jusjih, K.Nevelsteen, K.lee, KANURI SRINIVAS, Karlzt, Kbh3rd, Kdakin, Keilana, KellyCoinGuy, Kenny sh, Khalid hassani, Kongr43gpen, Kusunose, Kwekubo, Larry_Sanger, Laurens-af, Lev, Lfdder, Liberty Miller, Liempt, Lightmouse, Ligulem, Longhair, LuchoX, Lucky7654321, Lulu of the Lotus-Eaters, Luna Santin, M, MAG1, Mac, Madhero88, Maestro magico, Magister Mathematicae, Mani1, Manop, Martijn Hoekstra, MartinRe, Martynas Patasius, Marudubshinki, Matty4123, Maximaximax, Mayur, McGeddon, Mercer island student, Mermaid from the Baltic Sea, Metrax, Miguelfms, Mike Rosoft, Mike Van Emmerik, Mikrosam Akademija 2, Mild Bill Hiccup, Mindmatrix, Mlpkr, MmisNarifAlhoceimi, Mohamedsayad, Mortenoesterlundjoergensen, Murray Langton, Nanshu, Nickokillah, Nikai, Nixdorf, Noctibus, Noosentaal, NovaSTL, Ohnoitsjamie, Oicumayberight, Oliver Pereira, Onopearls, Orange Suede Sofa, OrgasGirl, Palnu, Paulkramer, Pearle, PetterBudt, Pharaoh of the Wizards, Philip Trueman, Poor Yorick, Power User, Proofreader77, Quota, Quuxplusone, R. S. Shaw, R. fiend, Racerx11, Radarjw, Radon210, Raise exception, Raven in Orbit, Rdsmith4, RedWolf, Rich Farmbrough, Rjwilmsi, Robert123R, Roybristow, Rusty Cashman, Ruud Koot, S.Örvarr.S, Sadi Carnot, Sae1962, Sannse, Saros136, Satellizer, Sean.hoyland, Sebbb-m, Sfahey, Shanes, SigmaEpsilon, Silver hr, SimonD, Sir Anon, Sir Nicholas de Mimsy-Porpington, Sjö, Skizzik, SlackerMom, Slady, Slashem, Slowking Man, Smiller933, SqlPac, Stephenb, Stevertigo, Storm Rider, Subdolous, Suisui, Symphony Girl, TBloemink, TakuyaMurata, Template namespace initialisation script, Tgeairn, Tharkowitz, The Anome, The Thing That Should Not Be, TheTechnoKid, Thecheesykid, Thegreenflashlight, Thingg, Thumperward, TiagoTiago, Tide rolls, Timhowardriley, Timshelton114, Tobias Bergemann, Tobiasjwt, TomasBat, Tommy2010, TonyClarke, Tpbradbury, Troels Arvin, True Genius, Ukexpat, UrbanBard, VIKIPEDIA IS AN ANUS!, Vcelloho, WJetChao, Welsh, Wereon, Wernher, Wesley, WhatamIdoing, Wiki alf, WikiDan61, Wikijens, Wikiloop, Wolfkeeper, XJaM, Xn4, Xp54321, Yidisheryid, Yintan, Ykhwong, Yonaa, Zipircik, ZonkBB6, Zundark, Zzuuzz, 470 anonymous edits

Programming language *Source*: http://en.wikipedia.org/w/index.php?oldid=608959153 *Contributors*: -Barry-, 10.7, 151.203.224.xxx, 16@r, 198.97.55.xxx, 199.196.144.xxx, 1exec1, 203.37.81.xxx, 212.188.19.xxx, 2988, 96.186, A. Parrot, A.amitkumar, A520, AJim, Abednigo, Abeliavsky, Abram.carolan, Acacix, Acaciz, AccurateOne, Addicted2Sanity, Ahoerstemeier, Ahy1, Akadruid, Alansohn, Alex, AlexPlank, Alhade, Alhoori, Alksub, Allan McInnes, Alliswellthen, Altenmann, Amire80, Amitkumargarg88, AmosWolfe, Ancheta Wis, Andonic, Andre Engels, Andres, Andylmurphy, Angela, Angusmclellan, Antonielly, Ap, Apwestern, ArmadilloFromHell, AstroNomer, Asukite, Autocratique, Avono, B4hand, Behnam, Beland, Ben Ben, Ben Standeven, Benjaminct, Bevo, Bgwhite, Bh3u4m, BigSmoke, Bill122, BioPupil, BirgitteSB, Blaisorblade, Blanchardb, Bobblewik, Bobbygammill, Bobo192, Bonaovox, Booyabazooka, Borislav, Brandon, Brentdax, Brianjd, Brick Thrower, Brion VIBBER, Bubba73, Burkedavis, CSProfBill, Calltech, Can't sleep, clown will eat me, CanisRufus, Capricorn42, Captain Conundrum, Captain-n00dle, CarlHewitt, Carmichael, Carrot Lord, Catgut, Cedar101, Centrx, Charlesriver, Charlie Huggard, Chillum, Chinabuffalo, Chun-hian, Cireshoe, Ckatz, Closedmouth, Cmdodanli, Cmichael, Cnilep, Cobaltbluetony, Cogiati, ColdFeet, Compfreak7, Conor123777, Conversion script, Cp15, Cybercobra, DBigXray, DMacks, DVD R W, Damian.rouson, Damieng, Dan128, Danakil, Danielmask, Danim, Dave Bell, David.Monniaux, DavidHOzAu, Davidfstr, Davidpdx, Dcoetzee, DeadEyeArrow, DenisMoskowitz, DennisDaniels, DerHexer, Derek Ross, Derek farn, Diannaa, Diego Moya, Dl2000, Dolfrog, Dominator09, Don't Copy That Floppy, Donhalcon, Doradus, DouglasGreen, DragonLord, Dreftymac, Dsimic, Dtaylor1984, Duke Ganote, Dysprosia, EJF, ESkog, EagleOne, Edward, Edward301, Eivind F Øyangen, ElAmericano, Elembis, EncMstr, EngineerScotty, Epbr123, Esap, Evercat, Everyking, Ewlyahoocom, Ezrakilty, Fantom, Faradayplank, Fastdrummer, Fayt82, Fieldday-sunday, Finlay McWalter, Fl, Foobah, Forderud, Four Dog Night, Fplay, Fraggle81, François Robere, Fredrik, Friedo, Fubar Obfusco, Funandtrvl, FvdP, Gaius Cornelius, Galoubet, Gazpacho, Gbruin, Georg Peter, Giftlite, Gilliam, Ginsuloft, Giorgios (usurped), Gioto, Goodgerster, Gploc, Green caterpillar, GregAsche, Grin, Grouphardev, Gurch, Gutza, Guy Harris, Gwicke, Hadal, Hairy Dude, Hammer1980, Hans Adler, HarisM, Harmil, Hayabusa future, Headbomb, HeikoEvermann, HenryLi, Hfastedge, HopeChrist, Hoziron, Hut 8.5, Hyad, INkubusse, IanOsgood, Icey, Ideogram, Ilario, Imran, Inaaaa, Indon, Infinoid, Iridescent, It Is Me Here, Iuliatoyo, Iwantitalllllllll, Ixfd64, J.delanoy, J991, JMSwtlk, JPINFV, JaK81600, JanSuchy, Jarble, Jason5ayers, Jaxad0127, Jaxl, Jeffrey Mall, Jeltz, Jeronimo, Jerryobject, Jesant13, Jguy, Jim1138, Jitse Niesen, Jj137, Johann Wolfgang, John lindgren, John254, JohnLai, JohnWittle, Jonesey95, Jonik, Jorend, Jossi, Joyous!, Jpbowen, Jpk, Jprg1966, Jschnur, JulesH, Juliancolton, Jusjih, Jwissick, K.lee, K12308025, KHaskell, KSmrq, KTC, Karingo, Karthiknd, Katieh5584, Kbdank71, Kbh3rd, Kedearian, Ketiltrout, Khalid Mahmood, Kickstart70, Kiminatheguardin, Kimse, Kinema, Klasbricks, KnowledgeOfSelf, Knyf, Komarov om, Kooginup, Koyaanis Qatsi, Kragen, Krauss, Krawi, Kris Schnee, Krischik, Kuciwalker, Kungfuadam, Kwertii, KymFarnik, L Gottschalk, L33tminion, LC, Lagalag, Leibniz, Liao, Lightmouse, Ligulem, LindsayH, LinguistAtLarge, Logarkh, LordCo Centre, Loriendrew, Lradrama, Lucian1900, Lucky7-phool, Lulu of the Lotus-Eaters, Luna Santin, Lupo, MER-C, MK8, Mac c, Macaldo, Macrakis, Magnus Manske, Mahanga, Majilis, Malcolm Farmer, Malleus Fatuorum, Mangojuice, Manpreett, Marcos, Mark Renier, MarsRover, MartinHarper, MartyMcGowan, Marudubshinki, Materialscientist, Matthew Woodcraft, Mattisse, Mav, Maxis ftw, McSly, Mccready, Mean as custard, MearsMan, MegaHasher, Mellum, Mendaliv, Merbabu, Merphant, Mesoderm, Michael Hardy, Michael Zimmermann, Midinastasurazz, Mike Rosoft, Mild Bill Hiccup, Minesweeper, MisterCharlie, Miym, Mkdw, Monz, Mpils, Mpradeep, Mrjeff, Ms2ger, Mschel, Muro de Aguas, Murray Langton, MusikAnimal, Mwaisberg, Mxn, Mithrandir, N5iln, Naderi 8189, Nameneko, Nanshu, Napi, Natalie Erin, Natkeeran, NawlinWiki, Nbarth, Nbrothers, Necklace, NewEnglandYankee, NewbieDoo, Nick125, Nikai, Nima1024, Ningauble, Nixdorf, Noctibus, Noformation, Noisy, Noldoaran, Noodleki, Noosentaal, NotQuiteEXPComplete, Nottsadol, Novasource, Ntalamai, Nuggetboy, Nutsnbolts222, Oblivious, Ohms law, Ohnoitsjamie, Oldadamml, Oleg Alexandrov, Olmerta, Omphaloscope, OrgasGirl, Orphan Wiki, Ottershrew, Papercutbiology, Paul August, PaulFord, Pcap, Peter, Peter Flass, Peterdjones, Pharaoh of the Wizards, Phil Sandifer, PhilKnight, Philg88, Photonique, Phyzome, Pieguy48, Piet Delport, PlayStation 69, Poor Yorick, Pooryorick, Positron, Prolog, Ptk, Pumpie, Pwv1, Quagmire, Quiddity, Quota, Quuxplusone, Qwyrxian, RainerBlome, Raise exception, Ranafon, RayAYang, RedWolf, Reddi, Reelrt, Reinis, RenamedUser2, Revived, RexNL, Rezonansowy, Rich Farmbrough, Rjstott, Rjwilmsi, Rlee0001, Robbe, Robert A West, Robert Skyhawk, Robo Cop, Roland2, Romanm, Ronhjones, Roux, Royboycrashfan, Rrburke, Rursus, Rushyo, Russell Joseph McCann, Ruud Koot, S.Örvarr.S, Saccade, Sam Korn, Science History, Seanhalle, Seaphoto, SeeAnd, Sekelsenmat, Sgbirch, Shadowjams, Shane A. Bender, Shanes, ShelfSkewed, SimonP, Simplyanil, Sjakkalle, Skytreader, Slaad, Slakr, Slashem, SmartBee, Snickel11, Sonicology, SparsityProblem, Specs112, Speed Air Man, SpeedyGonsales, Speuler, SpuriousQ, Steel1943, Stephen B Streater, Stephenb, Stevertigo, SubSeven, Suffusion of Yellow, Suruena, Swatiri, Swirsky, Switchercat, Systemetsys, TakuyaMurata, Tarret, Taxman, Teammm, Techman224, Tedickey, Template namespace initialisation script, Tentinator, Teval, Tewy, Tgeairn, Tgr, The Thing That Should Not Be, TheTechFan, Thetimperson, Thniels, Thomasuniko, Thumperward, Thv, Tiddly Tom, Tide rolls, Tim Starling, Timhowardriley, Tizio, Tobias Bergemann, TomT0m, Tomatensaft, Tony1, TonyClarke, Torc2, Toussaint, Trusilver, TuukkaH, Tysto, Ubiq, Ulric1313, Ultra two, Undeference, Useight, Usman&muzammal, Vadmium, Vahid83, Vaibhavkanwal, Vald, VampWillow, VictorAnyakin, Victorgrigas, Vivin, Vkhaitan, Vriullop, Vsion, WAS 4.250, Waterfles, Wavelength, Wiki alf, Wiki13, Wiki4Blog, WikiTome, Wikibob, Wikibofh, Wikisedia, Wimt, Windharp, Wlievens, Wmahan, Woohookitty, Ww, Xaosflux, Xavier Combelle, Yana209, Yath, Yk Yk Yk, Yoric, Zaheen, Zarniwoot, Zawersh, ZeWrestler, Zero1328, Zoicon5, Zondor, ²¹², Σ, ثاراس نوری, 1003 anonymous edits

Abstraction *Source*: http://en.wikipedia.org/w/index.php?oldid=605774945 *Contributors*: A520, Abednigo, AbstractTruth, Acdx, Aeons, Ahoerstemeier, Allan McInnes, Andre Engels, Andreas Kaufmann, Atoll, Benc, Bluemoose, Brad7777, C5st4wr6ch, CanisRufus, CarlHewitt, Causa sui, CharlesC, Codename Lisa, Cybercobra, Daclyff, David.Monniaux, Dawidl, Diego Moya, Dnas, Doug Bell, Dysprosia, Ellmist, Esap, Fgnievinski, FritzSolms, Gadfium, GateKeeper, Gilderien, Hans Dunkelberg, Hede2000, InYOUen0, JC Chu, JRR Trollkien, Jamelan, Jamesday, Jesse V., Jim1138, Jimpoz, JonHarder, Jpbowen, K.Nevelsteen, KeyStroke, Kuroi2014, Leafyplant, Libcub, Lighthill, Lotje, Luciole2013, MZMcBride, Maghnus, Maria Vargas, Mark Renier, Maurice Carbonaro, Mblumber, Michael Hardy, MikeSchinkel, Minesweeper, Mjb, Mohamed Magdy, Montalvo, Monural plural, Mordomo, Mwoolf, Node42, O, Oleg Alexandrov, Paulo Jorge Tomé, Pcap, Pjvpjv, Pnm, RRM, Ramu50, RedWolf, Rich Farmbrough, Rigadoun, Ru.spider, Sam Pointon, Sam Staton, SchreiberBike, Scott Ritchie, Sereine52, Shahid nx, Shawn wiki, Shervinafshar, Shimei, Slaggart, Snuffkin, Spiel496, TakuyaMurata, Template namespace initialisation script, TeunSpaans, The Anome, Thv, Tobias Bergemann, Torfason, Treygdor, TuukkaH, VictorAnyakin, Vonkje, Vroo, Wamiq, Wbm1058, Wik, Woohookitty, Zoicon5, 172 anonymous edits

Programmer *Source*: http://en.wikipedia.org/w/index.php?oldid=609111778 *Contributors*: 16@r, 217.98.151.xxx, 7, A Man In Black, Aan10akbar, Aatif.haider, Achraf52, AdjustShift, Adriaan, Adv94, AgentPeppermint, Akjar13, Alansohn, Aldaron, Alphaios, Alpunin, AndrewDressel, Antandrus, Arbsn, ArmadilloFromHell, ArnoldReinhold, Artemis.Fowl.Osborn, Ary29, Asa, Ata Sanderson, Azeri.programmer, AznAcorn, BananaFiend, Banej, Barticus88, Bdon190, Beland, Belovedfreak, Bevo, Billjoeme, Billpg, Blehfu, Bobrayner, Boofer908, Borgx, Bradtem, Brick Thrower, BrokenSegue, C.Fred, CONFIQ, CalendarWatcher, CalumH93, Cannolis, Carbonite, CasualVisitor, Cazzchazz, CesarB, Chocorisu, Clpo13, Conversion script, Curtis23, Cybercobra, DARTH SIDIOUS 2, DDerby, DEddy, DVD R W, DagErlingSmørgrav, Dahn, DancingPhilosopher, Dancter, Davewho2, David Biddulph, David Gerard, Davidruben, DeadEyeArrow, Denisarona, DerHexer, Dgmjr05, Diamondland, Dipankan001, Discospinster, Dylan620, Dysprosia, EIEIO, Eaglepuffs, EgyptAlex, Emperorbma, EncMstr, Epbr123, Epicgenius, Everyking, Evilcookies, Fadikaouk, Fidlej, Fissionfox, Fluzwup, Flyer22, Fraggle81, Frap, Frecklefoot, Fredrik, Galzigler, Geniac, Glane23, Gogo Dodo, Graham87, Greenrd, Greyhood, Gtdp, Gtstricky, Gwern, Gwernol, H, Hadal, Hajhouse, Hallows AG, Harris7, Harvester, Havrer0, Hcobb, Hekerui, HelgeStenstrom, Hellknowz, Hereticam, Hex539, Heymid, Hfc1875, Hollak4, Hu12, Iridescent, IronGargoyle, Ivan Štambuk, IvanLanin, Ixfd64, JForget, JLaTondre, JNW, Jaek112j, Jaka14, Jakezing, JamminBen, Jan1nad, Jarble, JasonMacker, Jatkins, Javierito92, Jay Gatsby, Jbsingh86, Jdpipe, JeR, Jed S, Jeff G., Jeh, Joaquin008, Joaquindesalva, Jojalozzo, Joncnunn, Jsharpminor, Jérôme, K.Nevelsteen, Kaare, Kbdank71, KeithH, KeithMatthew, Kenny sh, Kkm010, Konstable, Kostisl, Krashlandon, Ksnow, Kurniasan, Kyorosuke, L Kensington, Lalamax, LeadSongDog, Leafyplant, Lefteh, Leszek Jańczuk, Lightmouse, Liridon, Little Mountain 5, LittleOldMe, Lliberto, Longgg johnnn?, Loomisd, MER-C, Mabdul, Makled, Mark Renier, Matbeeche, Materialscientist, Matt Crypto, Matusz, Mekong Bluesman, Metalwario, Mgiganteus1, Michael Hardy, Michael Snow, Midgrid, Mike.lifeguard, Mild Bill Hiccup, Mindmatrix, Mountain, Moverton, MrOllie, Muckmed, MusikAnimal, Myork, Mythdon, NeonMerlin, NerdyNSK, Netoholic, Nikhilb239, Ohnoitsjamie, Oli Filth, Olly150, Omegamormegil, OrgasGirl, Ozonew4m, PMLawrence, Park3r, Pedant17, Peiman67, Perfectblue97, Petrb, Phantomsteve, Philip Trueman, Phillabust, Piet Delport, Piotrus, Prettydove, Prolog, Prunesqualer, Psyno, Pxtreme75, Quigabyte, Qute, R0pe-196, RA0808, Radarjw, Radiant!, RainbowOfLight, RandomAct, RandyGBrooks, Rdsmith4, Reedy, Rich Farmbrough, Richard R White, RobertG, Romanm, Rompe, Rongrongcheng, Ruud Koot, Rwwww, Saforrest, Sajisss, Salgueiro, Sam Hocevar, SciAndTech, Shafaet, Shoessss, SimonP, Sin Harvest, Sionus, Sjakkalle, Skittleys, Skylar Harrison, Slady, Slowking Man, Snoyes, Sooner Dave, Sorib, Starionwolf, Steel1943, Steevo714, Steven Zhang, Strmore, SyntaxError55, Tablizer, Tagishsimon, Taohinton, Tedickey, Terrek, The Grant Guy, Tim Starling, Time3000, Timotab, Tobby72, Tojge, TomasBat, Torc2, Torla42, Tpbradbury, Turkish soldier69, UR6LAD, Unforgettableid, User 38, Van helsing, Varundbest10, Veinor, Vendettax, Versageek, Waiwai933, Wiki alf, WikiMichel, Wikidenizen, Wikiklrsc, Wikipelli, XCC2, Xiaomao123, Yaron K., Yiliang, Yworo, Zayani, Zondor, Zvar, Σ, 483 anonymous edits

Language primitive *Source*: http://en.wikipedia.org/w/index.php?oldid=586895716 *Contributors*: A Stop at Willoughby, Andreas Kaufmann, Bhny, ChrisGualtieri, Diego Moya, Giraffedata, GoingBatty, Pcap, R'n'B, SchreiberBike, Woohookitty, 6 anonymous edits

Assembly language *Source*: http://en.wikipedia.org/w/index.php?oldid=608280492 *Contributors*: 16@r, 212.188.19.xxx, 28bytes, 3Nigma, 594rk, A.R., AMRAN AL-AGHBARI, Abdull, Accatagon, Ahoerstemeier, Ahy1, Aitias, Akamad, Akyprian, Alan d, Ale jrb, Aleph Infinity, Alex.g, Alfio, An-chan, Andonic, Andre Engels, Andres, Andy Dingley, Angela, Anger22, Angusus, Anna Lincoln, AnnaFrance, Another-anomaly, AnthonyQBachler, Anwar saadat, Ap, Apokrif, Arbabarehman, ArchiSchmedes, Areebmajeed, Ashmoo, Athaenara, Atlant, Audriusa, Autodmc, Backslash Forwardslash, Beefball, Beland, Ben Moore, Bender235, Beno1000, Bevo, Bfitzh2o, Bigdumbdinosaur, Bilbo1507, BillyPreset, Binrapt, Bison bloke, Blainster, Blakegripling ph, Blashyrk, Brage, BroodKiller, Bryan Derksen, Bumm13, ButOnMethItIs, CRGreathouse, CUTKD, Can't sleep, clown will eat me, CanOfWorms, CanisRufus, Capi crimm, Capitalismojo, Casascius, Cedar101, Chasingsol, Chatul, ChazZeromus, Cheeselet, Chester Markel, Chieffancypants, Chopin1810, Chris Chittleborough, ChrisGualtieri, Christian List, Cognoscent, Colejohnson66, Compfreak7, Conversion script, Coolv, Cpiral, Cquarksnow, Crotalus horridus, CultureDrone, DMacks, Damian Yerrick, Daniel Santos, Darktemplar, Darrien, Dasnov, DavidCary, Dawkcodave, Dcoetzee, Dead Horsey, DeanHarding, DekuDekuplex, Denispir, Derek Ross, Derek farn, Dexter Nextnumber, Discospinster, Dmbrunton, Doug, DouglasGreen, Dragon DASH, Drlegendre, Dsavi.x4, Dtgm, Eagle246, Eagleal, Easyas12c, Edward, Edward.in.Edmonton, Eleolar, Elving, Emperorbma, Emuguru, Epbr123, EricR, Everyking, EvilKeyboardCat, Evilfishy1, FDD, FallingGravity, Feezo, Femto, Ferkelparade, Filemon, Flewis, Fmiddleton, Fraggle81, François Robere, Frap, Fred Bradstadt, Frosty, Furrykef, Fuzzbox, Gaius Cornelius, Galoubet, Galzigler, Gannimo, Garas, Geau, GermanX, Giddie, Giftlite, Giobe2000, Gioto, GoldenMeadows, Gondooley, Goodone121, Grafen, Graham87, Greenrd, Greensburger, Gutsul, Guy Harris, Gwern, Haiviet, Hans Dunkelberg, HenkeB, Henriok, Herzleid, Hirzel, Hoo man, Hotdogger, HumphreyW, Hvn0413, IanOsgood, Inzy, ItsProgrammable, Ixfd64, JaGa, Jack O exiled, Jacob.jose, James086, JamesBWatson, Jamesx12345, JavierMC, Jeffrey Mall, Jeh, Jerome Charles Potts, Jerryobject, Jesant13, Jiveshwar, JoeBruno, JohnCJWood, Jorgon, Jpsowin, Jth299, Jukrat, Just Another Dan, KD5TVI, KP-Adhikari, Karada, Kaster, Kbdank71, Kbrose, Kdakin, Keilana, Kglavin, Khunglongcon, Kindall, Kjp993, Koektrommel, Konstable, Kubanczyk, Kwamikagami, LOL, LeaveSleaves, LeeZ, LizardJr8, Lkopeter, Loadmaster, Locke Cole, Longtt89, Lousyd, Lowellian, MCWNT, MER-C, MP64, Macofe, Manatee0, Marasmusine, Mario Blunk, Martarius, Martynas Patasius, Masgatotkaca, MasterProf, Materialscientist, Matt B., Mcandre, Mdanh2002, Mdwh, Meera123, Mellamokb, Mellum, Mendalus, Michal Jurosz, Michele.alessandrini, Miguelito Vieira, Mike Field, Mikellewellyn, Mindmatrix, Mkouklis, Mmernex, MonstaPro, Monz, Moonlit Knight, Mozillaman, Mr.Do!, MrOllie, Msikma, Mtpaley, Murray Langton, Mustafazamany, Mykk, NERIUM, Nandesuka, Nanshu, Nigelj, NightFalcon90909, Nikai, Nmcclana, Nusumareta, Ohnoitsjamie, Okane, OldCodger2, Oldhamlet,

Ospix, Owl3638, PJonDevelopment, PamD, Paresthesias, Patrick, Paul August, Pcap, Peetz1, Peter Flass, Pgk, Piano non troppo, Pinethicket, Pingveno, Pneuhaus, Pnm, Pol098, Poolisfun, Popsracer, Praefectorian, Quadell, Quarryman, R. S. Shaw, RCX, RHaworth, Ramu50, Rasmus Faber, Rbakels, Rdnk, Reaper Eternal, Red Prince, Regancy42, Reinderien, RexNL, Rezonansowy, Rfc1394, Rich Farmbrough, Rivanov, Rjwilmsi, Robbe, Robert Merkel, Ronz, Rotundo, Ruud Koot, Sanbec, Sanoj1234, Sanpnr, Schultkl, Scientus, Scipius, Scott Gall, Shadanan, Shadow demon, Shadowjams, SilentC, Simon80, SimonP, SkyWalker, Slaryn, Slashme, Sleigh, Slightsmile, Solidpoint, Soumyasch, Spalding, SpareHeadOne, SpeedyGonsales, Spinality, Sploonie, SpooK, SpuriousQ, Srice13, Starionwolf, Starnestommy, Startswithj, StealthFox, SteinbDJ, Stewartadcock, Stmrlbs, Stormy Ordos, Struway, Stuart Morrow, Subversive.sound, Superm401, Surturz, Suruena, Swtpc6800, Syjsoc, System86, TParis, Tarikes, Tbhotch, Tcsetattr, TeaDrinker, Tedickey, Teply, Th1rt3en, The Editors United, The Thing That Should Not Be, TheStarman, ThomasHarte, Ththrthtrh, Tide rolls, Tim32, Tobias Bergemann, Toksyuryel, Tomasz Tybulewicz, Tonymec, Toussaint, Trijnstel, True Pagan Warrior, Trusilver, Tsetsee yugi, Tusonchaz, Tzarius, Uli, Ultimus, Utvik, VampWillow, Vanished user 9i39j3, Vanished user ikijeirw34iuaeolaseriffic, Vegaswikian, Velle, Versus22, Vid512, Vobis132, Vwollan, Wavelength, Wengier, Wereon, Wernher, Wesley, Whitehatnetizen, Wiki alf, Wikiklrsc, Wilky DiFendare, Wizardman, Wjl2, Wknight8111, Wolfmankurd, Wre2wre, Wrp103, Wtshymanski, Wwmbes, XJaM, Xymmax, Ysangkok, Yworo, Zarel, Zonination, Zundark, Zx-man, ZyMOS, 810 anonymous edits

Machine code *Source*: http://en.wikipedia.org/w/index.php?oldid=606726190 *Contributors*: .:Ajvol:., 10metreh, 16@r, 192.35.241.xxx, 6a4fe8aa039615ebd9ddb83d6acf9a1dc1b684f7, AKismet, AdmN, AgadaUrbanit, Aguinaldo, Alfio, Algebra, Altenmann, Andre Engels, Beland, BiT, Bigdumbdinosaur, Bnugia, Burnishe, CanisRufus, Causa sui, CharlesC, Chester Markel, Cjewell, Cmdrjameson, Coinmanj, Conversion script, Cst17, Cyan.aqua, DJ Clayworth, DMacks, DarkShroom, Darkchoc4, Dav4is, Dawnseeker2000, Derek Andrews, Dgw, Discospinster, Donner60, Dori, El C, Eric-Wester, Everyking, Fabrictramp, Faizan, Feezo, François Robere, FrederikHertzum, Furrykef, G0gogcsc300, Galzigler, Georg Peter, GeorgeLouis, HLachman, Hans Dunkelberg, Helix84, HenkeB, Husond, Imjustmatthew, Inaaaa, Incnis Mrsi, Instinct, Ipsign, IronInforcer, Isarra, IvanLanin, J.delanoy, JDP90, Jacob.jose, Jagdeepyadav, Javert, Jeff02, Jesant13, Jesdisciple, Jeshan, Johnny 0, JonHarder, Jpk, Jusjih, Kappa, Karol Langner, Katieh5584, Kbdank71, Kim Bruning, Klaser, Kvng, LazyEditor, Lee.crabtree, Lekrecteurmasque, LiDaobing, LilHelpa, Lowellian, Lyricmac, Manop, Mark, Mark viking, Mastergreg82, Meadowbert, Mega Chrome, Megatronium, Melchoir, Microprofessor, Midzata, Mike Van Emmerik, Mindmatrix, Mirror Vax, MmisNarifAlhoceimi, Modest Genius, Muad, Murray Langton, Mxn, Nanshu, NawlinWiki, Nikai, OrgasGirl, Ost316, PBS, Pantergraph, Paul Stansifer, Prolog, Punctilius, QofASpiewak, Quuxplusone, Randomguy121, Reatlas, Rocketrod1960, Romanm, RussBlau, Rwpostiii, Salsa Shark, Sam Vimes, Sannse, Shmuel, Simoneau, Sky Harbor, Slady, Slipstream, Smalljim, Soumyasch, Spiritia, Stmrlbs, Sychen, TPIRFanSteve, Tedp, The Magician, Tisane, Tobias Bergemann, Toussaint, Tximist, UnfriendlyFire, VKokielov, Waggers, Walk&check, Wayfarer, WikiBully, Wikinerd, Wtshymanski, Xihix, Xod, Yanco, Yunshui, ZedaPhi, రవీందర్, 255 anonymous edits

Source code *Source*: http://en.wikipedia.org/w/index.php?oldid=607815799 *Contributors*: -Barry-, 0x6D667061, 16@r, Abc 123 def 456, Ablonus, Adam majewski, Aeolien, Ahy1, Alansohn, AlimanRuna, Allens, AnOddName, Anaraug, Andres, Angosso.com1, Ann O'nyme, AstroNomer, Barf73, Beland, Betacommand, Betterworld, Bevo, Bharatkumar, Brandalone, Bstnz, BurnDownBabylon, C17GMaster, CO, CYD, Calbaer, Can't sleep, clown will eat me, CanisRufus, CesarB, Chaojoker, CharlesC, Chris Pickett, Christiancatchpole, Chuunen Baka, Cleardelta, Conan, Conversion script, DC, Danakil, Darksun, Darkwind, DavidCary, Demonkoryu, DerHexer, Derek farn, Devourer09, Diego Moya, Dillard421, Discospinster, Doc glasgow, Domenico De Felice, Drano, Drdick, Dream Focus, Dreftymac, Dysprosia, ESkog, Ed Poor, EduardoCruz, Edward, Elcriz000, Epbr123, Erkan Yilmaz, Excirial, Faller, Ferengi, Fram, Frazzydee, FrenchIsAwesome, G0gogcsc300, GRAHAMUK, Gabeedwards, Gailtb, Galoubet, Gary, Gayathri prl, Giftlite, Ginsuloft, Grandscribe, GrayFullbuster, Green Tentacle, Greensburger, Gustavb, Gutworth, HYH.124, Haakon, Habj, Hans Dunkelberg, Harald Hansen, Hazard-SJ, Heron, Hooperbloob, Hu12, IRP, Ictlogist, Inaaaa, Incnis Mrsi, Inonit, Instigate cjsc (Narine), InverseHypercube, Iridescent, IvanLanin, J.delanoy, JLaTondre, JNW, Jackbrear, Jauhienij, Jay-Sebastos, Jeremy Visser, Jesse V., Jjalexand, Jjc259, Jni, Jsled, Kaare, Karlzt, Katieh5584, Kenny sh, Kop, Lantay77, LeaveSleaves, Longhair, Lord Anubis, Lotje, Lowellian, Lugia2453, M, MONGO, Mac, Maelnuneb, Manop, Mark viking, Martarius, Marudubshinki, Materialscientist, Mav, Mentifisto, Mesoderm, Midnightcomm, Mike Van Emmerik, Mindmatrix, Minimac's Clone, Mononomic, Mormegil, Mwtoews, Mxn, My name is not dave, Namazu-tron, Nanshu, NetRolller 3D, Neutrality, Nickj, Nova77, Nurg, Ohnoitsjamie, Onearc, Ormers, Oxymoron83, Pallas44, Papa November, Patrick, Paul Stansifer, Pdcook, Peter513414, Philip Trueman, Plrk, Ptrb, Pufferfish101, Pyrospirit, Quasipalm, Qwertyus, R Lowry, RavennaMoeba, Recognizance, Rettetast, Rholton, Rjaf29, Rob Hooft, Robert Merkel, Roboshed, RockMFR, Romanm, Romanski, RossPatterson, RoyBoy, Rp, Rrelf, Sanchezluis2020, SchnitzelMannGreek, Schzmo, Scott, Sean 1996, Shreevatsa, Sibian, Simetrical, Star767, Stewartadcock, Synergy, Taeshadow, Tassedethe, Taw, Tdavies24, Technopat, Techtonik, Tegel, Thumperward, Tiddly Tom, Timwi, Tobias Bergemann, TomasBat, Tompsci, TotoBaggins, Ttudor, Turlo Lomon, Turnstep, Tysto, Ultimus, Unyoyega, Uriyan, Vancouver Outlaw, Vgranucci, VictorAnyakin, Virtualphtn, Voomoo, Wapcaplet, Wbm1058, Wendy Ferguson, WhisperToMe, Wikibofh, WikipedianMarlith, Wikisaver62, Wiooiw, Yahia.barie, Yaris678, Yuejia, Yug1rt, Zfr, Zhaladshar, Zomglolwtfzor, Zop1997, Zzuuzz, ماني, 刘, 344 anonymous edits

Command *Source*: http://en.wikipedia.org/w/index.php?oldid=599042908 *Contributors*: 354d, Al Lemos, Amikeco, AndreasPraefcke, Andres, AnnaP, Astatine-210, Avinash7075, BiT, Bjankuloski06en, ChrisGualtieri, Christopherlin, Cpiral, DBigXray, DanielRigal, DiddyElliott, Dysprosia, Favonian, Fryed-peach, Georg Peter, Gerben1974, Ghettoblaster, Graham87, Gregbard, Harvest day fool, HendrixEesti, Incnis Mrsi, Jamelan, Jarble, Jim McKeeth, JonHarder, Josh the Nerd, Kim Bruning, Krallja, Mdsam2, Minnaert, Mxn, Patrick, Pinar, Pnm, Poulpy, Redvers, Rossami, Sethearls, Sonett72, TakuyaMurata, Thumperward, Tigga, Titodutta, Uncle G, Vadmium, Wbm1058, Ykhwong, ZeroOne, مالیا, 45 anonymous edits

Execution *Source*: http://en.wikipedia.org/w/index.php?oldid=608396494 *Contributors*: 16@r, AJim, Abdull, AbstractBeliefs, Akavel, Alansohn, Altenmann, Arthur Rubin, Biscuittin, Can't sleep, clown will eat me, Cndonovan, Diego Moya, Everyking, Excirial, Fluffernutter, François Robere, Green caterpillar, Greensburger, Isderion, J04n, JonHarder, Kbdank71, Kku, Koveras, Magioladitis, Marciam66, Marius, Maurice Carbonaro, Michael Hardy, NawlinWiki, Neutral current, Pooryorick, RedWolf, Rilak, Robin S, Rwwww, Sae1962, SebastianHelm, Shyland, Slipstream, TomT0m, Waldir, Wikipelli, Winterst, Ykhwong, 40 anonymous edits

Programming language theory *Source*: http://en.wikipedia.org/w/index.php?oldid=588108839 *Contributors*: Akmalzhon, Alan Moraes, Allan McInnes, Andy Dingley, Antonielly, AshtonBenson, Christopher Monsanto, Cogiati, Cybercobra, DagErlingSmørgrav, Denispir, Diego Moya, Dysepsion, Edgar181, EngineerScotty, Eumolpo, Fredrik, Galzigler, Gary, Headbomb, Hossein bardareh, Jrtayloriv, K.lee, LittleWink, MilerWhite, Miym, PMLawrence, Pcap, Pgr94, Quackor, R'n'B, Ruud Koot, SDC, Schmonyon, Seidenstud, Spayrard, Steven shaw, Tom Duff, Warlordwolf, Zophar1, 歡顏, 47 anonymous edits

Type system *Source*: http://en.wikipedia.org/w/index.php?oldid=607895987 *Contributors*: 121a0012, 1exec1, A3 nm, Aaron Rotenberg, Adavidb, Administration, AdrianLozano, Adrianwn, Agarwal1975, Ahy1, Aleksd, Allan McInnes, AllenDowney, Altenmann, Alterego, AnAbsolutelyOriginalUsername42, AnAj, Ancheta Wis, AngryBear, Anshee, Antonielly, Anuroop Sirothia, Anwar saadat, Ash211, Atreyu42, Audriusa, AutumnSnow, AvicAWB, Barabum, Beland, Blaisorblade, Bluemoose, Bob O'Bob, Bosmon, Bubba73, Caesura, Carlosayam, Cartsiltolt, Cedar101, CheesyPuffs144, ChorizoLasagna, Chridd, CiudadanoGlobal, Cjoev, Classicalecon, Clements, Cntras, Comatose51, Compfreak7, Connelly, Cornellier, Cpiral, Craigbeveridge, Cybercobra, Daira Hopwood, Damian Yerrick, Danakil, Daniel-Dane, Daniel.langdon, Darxus, DaveVoorhis, David Nicoson, Dcoetzee, Dearingj, Denny, DiscipleRaynes, Divinity76, Donhalcon, Doradus, Dougher, DouglasGreen, Dpv, Dreftymac, Droob, Drumheller, Dsimic, Dysprosia, Edaelon, Edward, Elaz85, Electricmuffin11, Elliot.jaffe, Emperorbma, EngineerScotty, Ennui93, Epolk, Eptified, Erc, Eric119, Erietveld, Esap, Eugeneiiim, Euyyn, Exigentsky, Ezrakilty, Feis-Kontrol, Fieldmethods, Fooblizoo, Foxxygirltamara, Fredrik, Fubar Obfusco, Funandtrvl, Furby100, Furrykef, GB fan, Gail, Gdr, Georg Peter, Gerweck, Gf uip, Ghettoblaster, Giftlite, Gmlk, Gracenotes, Grandscribe, Grauenwolf, GrindtXX, Guppyfinsoup, Gwern, Hairy Dude, Hammer, Harmil, Hdante, Headbomb, Hippietrail, Htmlapps, Iggymwangi, Ihope127, Imperator3733, Iridescent, JIP, JLaTondre, Jarble, Jason5ayers, Javawizard, Jbolden1517, Jeargle, Jeban, Jef-Infojef, Jen savage, Jerome Charles Potts, Jeronimo, Jerryobject, Jfire, Jleedev, John lindgren, John of Reading, Jon Awbrey, JonHarder, Jopincar, Joswig, Jrtayloriv, Julesd, K.lee, Karl Dickman, Karouri, Kbdank71, Kbrose, Ken Gallager, Ketil, Konklone, Krischik, Kupiakos, Kurniasan, LOL, Langec, Larry V, Lawpjc, Leibniz, LittleDan, LoStrangolatore, Lowellian, Lulu of the Lotus-Eaters, Maclary, Mange01, MarXidad, Marclevel3, Mark Renier, Markskilbeck, Martin Hampl, Martinship, Marudubshinki, MattGiuca, MattOConnor, Maximaximax, Maximilianklein, Mfc, Michael Slone, Michal Jurosz, Mikeblas, Mikon, Minesweeper, Mjamja, Mjb, Mmdoogie, Mmernex, Mnduong, Mokhov, Moonwolf14, MrBlueSky, Mshonle, NHSavage, Nabla, Neilc, Nightstallion, Norm mit, Norman Ramsey, Nuno Tavares, Officiallyover, OlEnglish, OriumX, OrthogonalFrog, OwenVersteeg, P00r, Paddy3118, Palmcluster, Patrick, Paul Richter, Pcap, Pedant17, Peepeedia, Pengo, Peterdjones, Pfeilspitze, Phil Boswell, Phorgan1, Pit, Pjb3, Pomoxis, Poor Yorick, Prodego, Qwertyus, Qwfp, R. S. Shaw, Raise exception, RandalSchwartz, RedWolf, Reinderien, Richardpianka, Riley Huntley, Rjwilmsi, Robykiwi, Rogper, Rookkey, Ross Fraser, Roybristow, Ruud Koot, SLi, SadaraX, Sae1962, Sagaciousuk, SashaMarievskaya, Scmerlin, Seanhalle, Seliopou, Servel333, Simeon, Simetrical, SimonP, Simoneau, Slaniel, Smeatish, Snoyes, Stevenj, Strake, Sun Creator, Suruena, Svick, Swift, Sykopomp, Tablizer, TakuyaMurata, Tasc, Teemu Leisti, That Guy, From That Show!, The Anome, TheNightFly, TheProgrammer, Therog1, Thincat, Thumperward, Tim Starling, Tim Watson, Tobias Bergemann, TomStuart, Torc2, TuukkaH, Twilsonb, Updatebjarni, Urhixidur, VampWillow, VictorAnyakin, VladimirReshetnikov, Washi, Wavelength, Wgunther, WhiteCat, Wiki.Tango.Foxtrot, Wjhonson, Wlievens, Wrp103, Ww, YahoKa, Yahya Abdal-Aziz, Yoric, Ysoldak, Zron, Σ, 402 anonymous edits

Strongly typed programming language *Source*: http://en.wikipedia.org/w/index.php?oldid=556816448 *Contributors*: Bananabruno, Batiste93, Biscuittin, CJ1992, Chris the speller, Cybercobra, Dekart, Epicgenius, Gerrit, Grauenwolf, Haruth, Hydrox, Jarble, Kullanari, Lexlex, Niceguyedc, Odedrim, Paddy3118, Phil Boswell, RedWolf, Rjwilmsi, Ruud Koot, SashaMarievskaya, Sszydelko, Strombrg, 28 anonymous edits

Weak typing *Source*: http://en.wikipedia.org/w/index.php?oldid=556813945 *Contributors*: Bananabruno, Batiste93, Biscuittin, CJ1992, Chris the speller, Cybercobra, Dekart, Epicgenius, Gerrit, Grauenwolf, Haruth, Hydrox, Jarble, Kullanari, Lexlex, Niceguyedc, Odedrim, Paddy3118, Phil Boswell, RedWolf, Rjwilmsi, Ruud Koot, SashaMarievskaya, Sszydelko, Strombrg, 28 anonymous edits

Syntax *Source*: http://en.wikipedia.org/w/index.php?oldid=598492976 *Contributors*: 28bytes, Aaron Rotenberg, Allan McInnes, Bjankuloski06en, BlakeCS, CRGreathouse, Cedar101, Chris the speller, Cleared as filed, Cpiral, Derek R Bullamore, Derek farn, Diego Moya, Dom96, Edward, FrenchIsAwesome, Fusion7, Gf uip, Hurmata, Iskander s, It Is Me Here, Jarble, Jeffreykegler, Johan1298, Lokentaren, Mblumber, MmisNarifAlhoceimi, Natkeeran, Nbarth, Octahedron80, Pcap, Pjposullivan, RCX, RobertL, SemanticMantis, Thiago Miotto Amaral, Tijfo098, TomasBat, Zron, 27 anonymous edits

Scripting language *Source*: http://en.wikipedia.org/w/index.php?oldid=608376582 *Contributors*: -Barry-, 156.153.254.xxx, 217.99.96.xxx, Aavviof, Abstudio, AdmN, Ae-a, Alansohn, Aldie, Alexia Death, Ancheta Wis, Andre Engels, Antandrus, Antonielly, Ap0ught, Arvindn, Assasin Joe, AxelBoldt, B3rt0h, Bcward, Be..anyone, Beetstra, Bender235, Betterusername, Bevo, Bgwhite, Bilalis, Billposer, Bizza123, Bkengland, Blehfu, Bmdavll, Bongwarrior, BonzoESC, Breawycker, Bstepno, Burschik, Buzgun, C.lekberg, Capitalismojo, Captain Conundrum, Ccacsmss,

Chasingsol, Chinju, Cic, Cindamuse, Cmertayak, Colonies Chris, Conversion script, Corn cheese, Corsairtux, Corvus, CrazyChemGuy, Cryoboy, DARTH SIDIOUS 2, DNewhall, Danhash, DanteEspinoza1989, Darth Panda, Dc987, Dckelly, Dcoetzee, Deathbuilder, Der Golem, Diego Moya, Digichoron, Discospinster, DmitTrix, Dollyd, DonToto, Dori, Dougher, Dr.queso, Dreamfly1024, Dreftymac, Dsmdgold, Duffman, Dungodung, Dycedarg, ED1T3R, Easwarno1, EdC, Edupedro, Ehheh, Elcidia, Elfguy, Elwikipedista, Eman2129, EmeryD, Emurphy42, Ericcnelson, Etu, Exert, Exidor, Fabartus, FatalError, Flockmeal, Fluffernutter, Fraggle81, Frecklefoot, Fredrik, Freeman45Fighter, Freshraisin, Friedo, Fubar Obfusco, Func, Fırat KÜÇÜK, Gaius Cornelius, Galoubet, Geary, Generic Player, Geniac, Ghettoblaster, Gilliam, GjeDeeR!, Gogo Dodo, Gokudo, Graham87, Grendelkhan, Grinchfizzlekrump, Grunt, Gryllida, Gwernol, HQCentral, Hadal, Hajhouse, Hans Adler, Hao2lian, Harmil, Hassanbird, Hazel77, Helpersatan, HenkvD, Henry hedden, Hexacoder, Hobophobe, Hu12, HurricaneSpin, Hyperlink, IanManka, Immsubodh, Interiot, Israel Walker, Ivan Pozdeev, IvanLanin, JLaTondre, JMJimmy, JRocketeer, Jarble, Jcdietz03, Jcuevas, Jeenuv, Jeltz, Jerome Charles Potts, Jesant13, Jim Douglas, Jk2q3jrklse, Joeblakesley, Jonik, Jordancpeterson, Jsnover, Jwink3101, K.Nevelsteen, Karada, Karl Dickman, Keilana, Kenyon, Khalid hassani, Kiamlaluno, Kim Bruning, Kiski7c5, Kjetil r, Komarov om, Krauss, Ksn, Kvdveer, Kwi, L Kensington, Ledgerbob, Leujohn, Logan, Lorenzarius, Lrenh, Luk, LuoShengli, MER-C, MadSurgeon, Makecat, Marco Krohn, Mark Foskey, Mark Renier, Marqueed, Martyulrich, Marudubshinki, Maslen, Masterdeath01, Materialscientist, Matt Deres, Matusz, Mav, Maxim, Maximaximax, Mgr, Mike92591, Miles, Minghong, Minna Sora no Shita, Mipadi, MontyB, Moondyne, Mr. Wheely Guy, MrH, MrOllie, Mrzaius, Nanshu, NawlinWiki, Nbarth, Neilc, NellieBly, Nepenthes, NerdyNSK, Nickjames90, Night day, Nigosh, Niqueco, Nitin jeewan, Noamatari, NurAzije, OMPIRE, Olathe, OlavN, Olopez, Oosh, OracleGuy01, Orange Suede Sofa, Orb4peace, Ottershrew, PMDrive1061, Pavel Vozenilek, Peter Flass, Philip Trueman, Phillsmith01, Piet Delport, Pol098, Policechiefronaldo, PonThePony, Ppp extr, Pratyya Ghosh, Prestonmag, ProgJones, RadioFan, Raffaele Megabyte, Raghith, Ramir, Random account 47, Ranjithsutari, Raybob95, RenamedUser01302013, Rich Farmbrough, RichardOSmith, Ricvelozo, RobertL, Rowfilter, Rp, Ryzol, S0aasdf2sf, SMC, Satori, Sbasith, Scarlet, Scorwin, Shlomif, Shlomital, Simetrical, SimonEast, SimonP, Siodhe, Sjbrown, Skizzik, SkyWalker, Sligocki, SoCalSuperEagle, Softarch Jon, Spalding, Srushe, Startswithj, Stefan Urbanek, Stevietheman, Style, T Long, TOReilly, TShilo12, Tagishsimon, Tarquin, Tekeek, Teltek, TerezaS, TexasAndroid, Tfgbd, Thane, Thapthim, The High Magus, Thomas Veil, TianzhouChen, Tic260, TimoMax, Tobias Bergemann, Todobo, Tontito, Tony Sidaway, Toussaint, TowerDragon, Travis Evans, Tuntable, Turnstep, Twas Now, Two Bananas, Twsx, Tyw7, Uhai, Uriyan, User86654, VKokielov, Vampireq, Verloren, Visage, Waldir, WalterGR, Warren, Wermlandsdata, Wernher, Widr, Wiki alf, Wilfrednilsen, Wlievens, WookieInHeat, Wootery, Writers Bond, Writtenonsand, Xioxox, Xjas05, Xtremejames183, Yaron K., Yunshui, Yzt, ZX81, Zeimusu, Zuixro, Zzuuzz, مانی, 639 anonymous edits

Image Sources, Licenses and Contributors

file:Coding Shots Annual Plan high res-5.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Coding_Shots_Annual_Plan_high_res-5.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Matthew (WMF)

File:Ada lovelace.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Ada_lovelace.jpg *License*: Public Domain *Contributors*: Aavindraa, Coyau, Dcoetzee, DutchHoratius, Kaldari, Kelson, Kilom691, Michael Barera, 1 anonymous edits

File:PunchCardDecks.agr.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PunchCardDecks.agr.jpg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: mehul panchal

File:IBM402plugboard.Shrigley.wireside.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:IBM402plugboard.Shrigley.wireside.jpg *License*: Creative Commons Attribution 2.5 *Contributors*: User:ArnoldReinhold

File:H96566k.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:H96566k.jpg *License*: Public Domain *Contributors*: Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.

File:Object-Oriented-Programming-Methods-And-Classes-with-Inheritance.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Object-Oriented-Programming-Methods-And-Classes-with-Inheritance.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Carrot Lord

File:USB flash drive.JPG *Source*: http://en.wikipedia.org/w/index.php?title=File:USB_flash_drive.JPG *License*: GNU Free Documentation License *Contributors*: User:Nrbelex

File:Dg-nova3.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Dg-nova3.jpg *License*: Copyrighted free use *Contributors*: User Qu1j0t3 on en.wikipedia

File:Classes and Methods.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Classes_and_Methods.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Bobbygammill

File:Bangalore India Tech books for sale IMG 5261.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Bangalore_India_Tech_books_for_sale_IMG_5261.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Victorgrigas

Image:Python add5 parse.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Python_add5_parse.png *License*: Public Domain *Contributors*: User:Lulu of the Lotus-Eaters

Image:Python add5 syntax.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Python_add5_syntax.svg *License*: Copyrighted free use *Contributors*: Xander89

Image:Data abstraction levels.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Data_abstraction_levels.png *License*: Public Domain *Contributors*: Doug Bell, Perey

File:Bundesarchiv B 145 Bild-F031434-0006, Aachen, Technische Hochschule, Rechenzentrum.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Bundesarchiv_B_145_Bild-F031434-0006,_Aachen,_Technische_Hochschule,_Rechenzentrum.jpg *License*: Creative Commons Attribution-Sharealike 3.0 Germany *Contributors*: ArnoldReinhold, Martin H., YMS

Image:Ada Lovelace portrait.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Ada_Lovelace_portrait.jpg *License*: Public Domain *Contributors*: Jean-Frédéric, Jkadavoor, Julia W, Kaldari, Mindmatrix, Mywood, Pine, Piotrus, Shir-El too, SirHenryNorris, Tokorokoko, 1 anonymous edits

File:Офис Яндекса работа.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Офис_Яндекса_работа.jpg *License*: Creative Commons Attribution 2.0 *Contributors*: Alpunin, Kaganer

File:Motorola 6800 Assembly Language.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Motorola_6800_Assembly_Language.png *License*: Public Domain *Contributors*: Swtpc6800 en:User:Swtpc6800 Michael Holley

File:W65C816S Machine Code Monitor.jpeg *Source*: http://en.wikipedia.org/w/index.php?title=File:W65C816S_Machine_Code_Monitor.jpeg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Bigdumbdinosaur

Image:CodeCmmt002.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:CodeCmmt002.svg *License*: GNU Free Documentation License *Contributors*: Original uploader was Dreftymac at en.wikipedia

File:Lambda lc.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Lambda_lc.svg *License*: Public Domain *Contributors*: Cathy Richards, Luks, Vlsergey, 2 anonymous edits

Image:Python add5 parse.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Python_add5_parse.svg *License*: Public Domain *Contributors*: User:Lulu of the Lotus-Eaters

License