Wikipedia

Programming language

A **programming language** is any set of rules that converts strings, or graphical program elements in the case of visual programming languages, to various kinds of machine code output. Programming languages are one kind of computer language, and are used in computer programming to implement algorithms.

```c
/*
 * This line basically imports the "stdio" header file, part of
 * the standard library. It provides input and output functionality
 * to the program.
 */
#include <stdio.h>

/*
 * Function (method) declaration. This outputs "Hello, world\n" to
 * standard output when invoked.
 */
void sayHello(void) {
    // printf() in C outputs the specified text (with optional
    // formatting options) when invoked.
    printf("Hello, world!\n");
}

/*
 * This is a "main function". The compiled program will run the code
 * defined here.
 */
int main(void)
{
    // Invoke the sayHello() function.
    sayHello();
    return 0;
}
```

The source code for a simple computer program written in the C programming language. The gray lines are comments that help explain the program to humans in a natural language. When compiled and run, it will give the output "Hello, world!".

Most programming languages consist of instructions for computers. There are programmable machines that use a set of specific instructions, rather than general programming languages. Prior to the invention of computers, programs were used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos.[1]

Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form (i.e., as a sequence of operations to perform) while other languages use 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), which are usually defined by a formal language. 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. Some languages have both, with the basic language defined by a standard and extensions taken from the dominant implementation being common.

Programming language theory is a subfield of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages.

Contents

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm.[2] Some authors restrict the term "programming language" to those languages that can express all possible algorithms.[2][3] 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 involves a computer performing some kind of computation[4] or algorithm and possibly control external devices such as printers, disk drives, robots,[5] 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.[6] In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way.[7] 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.[8] This principle is sometimes formulated as a recommendation to the programmer to make proper use of such abstractions.[9]

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 are often called programming languages.[10][11]

Markup languages like XML, HTML, or troff, which define structured data, are not usually considered programming languages.[12][13][14] Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete language entirely using XML syntax.[15][16][17] Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.[18][19]

The term *computer language* is sometimes used interchangeably with programming language.[20] 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.[21] Similarly, 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.[22]

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.[23] 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.[24]

History

Early developments

Very early computers, such as Colossus, were programmed without the help of a stored program, by modifying their circuitry or setting banks of physical controls.

Slightly later, programs could be written in machine language, where the programmer writes each instruction in a numeric form the hardware can execute directly. For example, the instruction to add the value in two memory locations might consist of 3 numbers: an "opcode" that selects the "add" operation, and two memory locations. The programs, in decimal or binary form, were read in from punched cards, paper tape, magnetic tape or toggled in on switches on the front panel of the computer. Machine languages were later termed *first-generation programming languages* (1GL).

The next step was the development of the so-called *second-generation programming languages* (2GL) or assembly languages, which were still closely tied to the instruction set architecture of the specific computer. These served to make the program much more human-readable and relieved the programmer of tedious and error-prone address calculations.

The first *high-level programming languages*, or *third-generation programming languages* (3GL), 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.[25]

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer.[26] 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. As 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.[27][28]

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.

In 1954, FORTRAN was invented at IBM by John Backus. It was the first widely used high-level general purpose programming language to have a functional implementation, as opposed to just a design on paper.[29][30] It is still a popular language for high-performance computing[31] and is used for programs that benchmark and rank the world's fastest supercomputers.[32]

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.[33] The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959.[34] FLOW-MATIC was a major influence in the design of COBOL, since only it and its direct descendant AIMACO were in actual use at the time.[35]

Refinement

The increased use of high-level languages introduced a requirement for *low-level programming languages* or *system programming languages*. These languages, to varying degrees, provide facilities between assembly languages and high-level languages. They can be used to perform tasks that require direct access to hardware facilities but still provide higher-level control structures and error-checking.

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use:

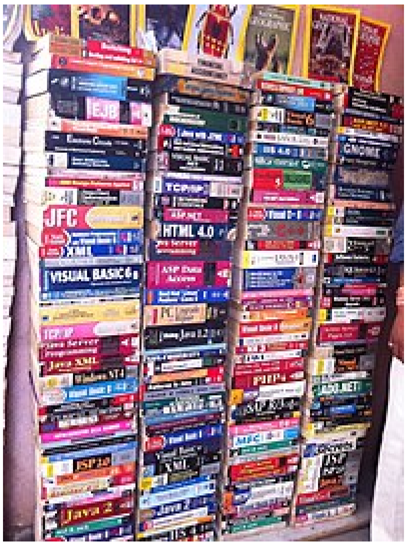
- APL introduced *array programming* and influenced functional programming.[36]
- ALGOL refined both *structured procedural programming* and the discipline of language specification; the "Revised Report on the Algorithmic Language ALGOL 60" became a model for how later language specifications were written.

- Lisp, implemented in 1958, was the first dynamically typed *functional programming* language.
- 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 for the Unix operating system and remains popular.[37]
- 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 descendants, and most modern programming 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.[38] 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.[39]

Consolidation and growth



A small selection of programming language textbooks

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 the so-called "fifth-generation" languages that incorporated logic programming constructs.[40] 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 decades.

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, which were often wedded to generic programming constructs.[41]

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 of many existing languages and paradigms (although their syntax was often 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.

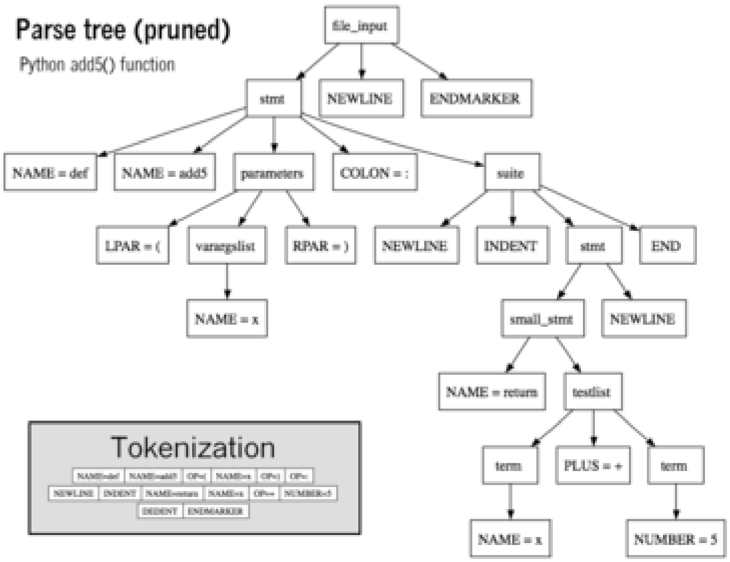
Fourth-generation programming languages (4GL) are computer programming languages that aim to provide a higher level of abstraction of the internal computer hardware details than 3GLs. *Fifth-generation programming languages* (5GL) are programming languages based on solving problems using constraints given to the program, rather than using an algorithm written by a programmer.

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

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.



Parse tree of Python code with inset tokenization

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.

```
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 n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '    %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

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

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:

```
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 undefined variable `p` during compilation. However, 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.[42] 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.[43] 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.[44]

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.[2] 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.[45] 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

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

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.[46] For example, the data represented by `"this text between the quotes"` is a string, and in many programming languages dividing a number by a string has no meaning and will not be executed. The invalid operation may be detected when the program is compiled ("static" type checking) and will be rejected by the compiler with a compilation error message, or it may be detected while the program is running ("dynamic" type checking), resulting in a run-time exception. Many languages allow a function called an exception handler to handle this exception and, for example, always return "-1" as the result.

A special case of typed languages are the *single-typed* 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, generally sequences of bits of various lengths.[46] High-level untyped languages include BCPL, Tcl, and some varieties of Forth.

In practice, while few languages are considered typed from the type theory (verifying or rejecting all operations), most modern languages offer a degree of typing.[46] 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 vis-à-vis 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.[46]

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, C++, Java and C# all infer types in certain limited cases.[47] Additionally, some programming languages allow for some types to be automatically converted to other types; for example, an int can be used where the program expects a float.

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*.[46] 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, Smalltalk, Perl, Python, JavaScript, and Ruby are all examples of dynamically typed languages.

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.[46] 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 these program faults. An attempt to perform an operation on the wrong type of value raises an error.[46] 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.[48][49]

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

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.

The line between a language and its core library differs from language to language. In some cases, the language designers may treat the library as a separate entity from the language. However, a language's core library is often treated as part of the language by its users, and some language specifications even require that this library be made available in all implementations. 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.[50][51] 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.[52] 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.[53] 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.
- 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.[54]

Natural language programming has been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant 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".[55] Alan Perlis was similarly dismissive of the idea.[56] 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

The specification of a programming language is an artifact that the language users and the implementors can use to agree upon whether a piece of source code is a valid program in that language, and if so what its behavior shall be.

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[57] and Scheme[58] 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[59]). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An *implementation* of a programming language provides a way to write programs in that language and execute them 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 much faster than those that are interpreted in software.[60]

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.

Proprietary languages

Although most of the most commonly used programming languages have fully open specifications and implementations, many programming languages exist only as proprietary programming languages with the implementation available only from a single vendor, which may claim that such a proprietary language is their intellectual property. Proprietary programming languages are commonly domain specific languages or internal scripting languages for a single product; some proprietary languages are used only internally within a vendor, while others are available to external users.

Some programming languages exist on the border between proprietary and open; for example, Oracle Corporation asserts proprietary rights to some aspects of the Java programming language,[61] and Microsoft's C# programming language, which has open implementations of most parts of the system, also has Common Language Runtime (CLR) as a closed environment.[62]

Many proprietary languages are widely used, in spite of their proprietary nature; examples include MATLAB, VBScript, and Wolfram Language. Some languages may make the transition from closed to open; for example, Erlang was originally an Ericsson's internal programming language.[63]

Use

Thousands of different programming languages have been created, mainly in the computing field.[64] Individual software projects commonly use five programming languages or more.[65]

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).[66] *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 can run its commands through an interpreter (such as a Unix shell or other command-line interface), without compiling, it is called a scripting language.[67]

Measuring language usage

Determining which is the most widely used programming language is difficult since the definition of usage varies by context. One language may occupy the greater number of programmer hours, a different one has more lines of code, and a third may consume 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;[68][69] Fortran in scientific and engineering applications; Ada in aerospace, transportation, military, real-time and embedded 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[70]
- the number of books sold that teach or describe the language[71]
- estimates of the number of existing lines of code written in the language – which may underestimate languages not often found in public searches[72]
- 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, stackify.com reported the ten most popular programming languages as (in descending order by overall popularity): Java, C, C++, Python, C#, JavaScript, VB .NET, R, PHP, and MATLAB.[73]

Dialects, flavors and implementations

A **dialect** of a programming language or a data exchange language is a (relatively small) variation or extension of the language that does not change its intrinsic nature. With languages such as Scheme and Forth, standards may be considered insufficient, inadequate or illegitimate by implementors, so often they will deviate from the standard, making a new dialect. In other cases, a dialect is created for use in a domain-specific language, often a subset. In the Lisp world, most languages that use basic S-expression syntax and Lisp-like semantics are considered Lisp dialects, although they vary wildly, as do, say, Racket and Clojure. As it is common for one language to have several dialects, it can become quite difficult for an inexperienced programmer to find the right documentation. The BASIC programming language has many dialects.

Taxonomies

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.[74] 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).[75] Some general purpose languages were designed largely with educational goals.[76]

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.

See also

- Comparison of programming languages (basic instructions)
- Comparison of programming languages
- Computer programming
- Computer science and Outline of computer science
- Domain-specific language
- Domain-specific modelling
- Educational programming language
- Esoteric programming language
- Extensible programming
- Category:Extensible syntax programming languages
- Invariant based programming
- List of BASIC dialects
- Lists of programming languages
- List of programming language researchers
- Programming languages used in most popular websites
- Language-oriented programming
- Logic programming
- Literate programming
- Metaprogramming
  - Ruby (programming language) § Metaprogramming
- Modeling language
- Programming language theory
- Pseudocode
- Rebol § Dialects
- Reflection
- Scientific programming language
- Scripting language
- Software engineering and List of software engineering topics

References

1. Ettinger, James (2004) *Jacquard's Web*, Oxford University Press
2. Aaby, Anthony (2004). *Introduction to Programming Languages* (https://web.archive.org/web/20121108043216/http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm). Archived from the original (http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm) on 8 November 2012. Retrieved 29 September 2012.
3. In mathematical terms, this means the programming language is Turing-complete MacLennan, Bruce J. (1987). *Principles of Programming Languages*. Oxford University Press. p. 1. ISBN 978-0-19-511306-8.
4. ACM SIGPLAN (2003). "Bylaws of the Special Interest Group on Programming Languages of the Association for Computing Machinery" (http://www.acm.org/sigs/sigplan/sigplan_bylaws.htm). Archived (https://web.archive.org/web/20060622110145/http://www.acm.org/sigs/sigplan/sigplan_bylaws.htm) from the original on 22 June 2006., "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."
5. Dean, Tom (2002). "Programming Robots" (http://www.cs.brown.edu/people/tld/courses/cs148/02/programming.html). *Building Intelligent Robots*. Brown University Department of Computer Science. Archived (https://web.archive.org/web/20061029045949/http://www.cs.brown.edu/people/tld/courses/cs148/02/programming.html) from the original on 29 October 2006.
6. 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 0-12-012108-5, 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]
7. Ben Ari, Mordechai (1996). *Understanding Programming Languages*. John Wiley and Sons. "Programs and languages can be defined as purely formal mathematical objects. However, more people are interested in programs than in other mathematical objects such as groups, precisely because it is possible to use the program—the sequence of symbols—to control the execution of a computer. While we highly recommend the study of the theory of programming, this text will generally limit itself to the study of programs as they are executed on a computer."
8. David A. Schmidt, *The structure of typed programming languages*, MIT Press, 1994, ISBN 0-262-19349-3, p. 32
9. Pierce, Benjamin (2002). *Types and Programming Languages* (https://archive.org/details/typesprogramming00pier_207). MIT Press. p. 339 (https://archive.org/details/typesprogramming00pier_207/page/n362). ISBN 978-0-262-16209-8.
10. Digital Equipment Corporation. "Information Technology – Database Language SQL (Proposed revised text of DIS 9075)" (http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt). *ISO/IEC 9075:1992, Database Language SQL*. Archived (https://web.archive.org/web/20060621035823/http://www.contrib.andrew.cmu.edu/%7Eshadow/sql/sql1992.txt) from the original on 21 June 2006. Retrieved 29 June 2006.
11. The Charity Development Group (December 1996). "The CHARITY Home Page" (http://pll.cpsc.ucalgary.ca/charity1/www/home.html). Archived (https://web.archive.org/web/20060718010551/http://pll.cpsc.ucalgary.ca/charity1/www/home.html) from the original on 18 July 2006., "Charity is a categorical programming language...", "All Charity computations terminate."

12. XML in 10 points (http://www.w3.org/XML/1999/XML-in-10-points.html) Archived (https://web.archive.org/web/20090906083110/http://www.w3.org/XML/1999/XML-in-10-points.html) 6 September 2009 at the Wayback Machine W3C, 1999, "XML is not a programming language."
13. Powell, Thomas (2003). *HTML & XHTML: the complete reference*. McGraw-Hill. p. 25. ISBN 978-0-07-222942-4. "HTML is not a programming language."
14. Dykes, Lucinda; Tittel, Ed (2005). *XML For Dummies* (https://archive.org/details/html4fordummies00titt_2) (4th ed.). Wiley. p. 20 (https://archive.org/details/html4fordummies00titt_2/page/20). ISBN 978-0-7645-8845-7. "...it's a markup language, not a programming language."
15. "What kind of language is XSLT?" (http://www.ibm.com/developerworks/library/x-xslt/). IBM.com. 20 April 2005. Archived (https://web.archive.org/web/20110511192712/http://www.ibm.com/developerworks/library/x-xslt/) from the original on 11 May 2011.
16. "XSLT is a Programming Language" (http://msdn.microsoft.com/en-us/library/ms767587(VS.85).aspx). Msdn.microsoft.com. Archived (https://web.archive.org/web/20110203015119/http://msdn.microsoft.com/en-us/library/ms767587(VS.85).aspx) from the original on 3 February 2011. Retrieved 3 December 2010.
17. Scott, Michael (2006). *Programming Language Pragmatics* (https://archive.org/details/programminglangu00scot_912). Morgan Kaufmann. p. 802 (https://archive.org/details/programminglangu00scot_912/page/n834). ISBN 978-0-12-633951-2. "XSLT, though highly specialized to the transformation of XML, is a Turing-complete programming language."
18. Oetiker, Tobias; Partl, Hubert; Hyna, Irene; Schlegl, Elisabeth (20 June 2016). "The Not So Short Introduction to LATEX 2ε" (https://tobi.oetiker.ch/lshort/lshort.pdf) (Version 5.06). *tobi.oetiker.ch*. pp. 1–157. Archived (https://web.archive.org/web/20170314015536/https://tobi.oetiker.ch/lshort/lshort.pdf) (PDF) from the original on 14 March 2017.
19. Syropoulos, Apostolos; Antonis Tsolomitis; Nick Sofroniou (2003). *Digital typography using LaTeX* (https://archive.org/details/digitaltypograph00syro_587). Springer-Verlag. p. 213 (https://archive.org/details/digitaltypograph00syro_587/page/n237). ISBN 978-0-387-95217-8. "TeX is not only an excellent typesetting engine but also a real programming language."
20. Robert A. Edmunds, The Prentice-Hall standard glossary of computer terminology, Prentice-Hall, 1985, p. 91
21. Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst, *Towards a General Ontology of Computer Programs (http://home.mis.u-picardie.fr/~site-ic/site/IMG/pdf/ICSOFT2007_final.pdf)* Archived (https://web.archive.org/web/20150707093557/http://home.mis.u-picardie.fr/~site-ic/site/IMG/pdf/ICSOFT2007_final.pdf) 7 July 2015 at the Wayback Machine, ICSOFT 2007 (http://dblp.uni-trier.de/db/conf/icsoft/icsoft2007-1.html) Archived (https://web.archive.org/web/20100427063709/http://dblp.uni-trier.de/db/conf/icsoft/icsoft2007-1.html) 27 April 2010 at the Wayback Machine, pp. 163–170
22. S.K. Bajpai, *Introduction To Computers And C Programming*, New Age International, 2007, ISBN 81-224-1379-X, p. 346
23. 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 0-12-012108-5, 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."

24. John C. Reynolds, "Some thoughts on teaching programming and programming languages", *SIGPLAN Notices*, Volume 43, Issue 11, November 2008, p.109
25. 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) Archived (https://web.archive.org/web/20141018204625/http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm) 18 October 2014 at the Wayback Machine
26. Sebesta, W.S Concepts of Programming languages. 2006;M6 14:18 pp.44. ISBN 0-321-33025-0
27. Knuth, Donald E.; Pardo, Luis Trabb. "Early development of programming languages". *Encyclopedia of Computer Science and Technology*. **7**: 419–493.
28. Peter J. Bentley (2012). *Digitized: The Science of Computers and how it Shapes Our World* (https://books.google.com/books?id=kpYX_INI0VMC). Oxford University Press. p. 87. ISBN 9780199693795. Archived (https://web.archive.org/web/20160829191955/https://books.google.com/books?id=kpYX_INI0VMC) from the original on 29 August 2016.
29. "Fortran creator John Backus dies – Tech and gadgets" (http://www.nbcnews.com/id/17704662). NBC News. 20 March 2007. Retrieved 25 April 2010.
30. "CSC-302 99S : Class 02: A Brief History of Programming Languages" (http://www.math.grin.edu/~rebelsky/Courses/CS302/99S/Outlines/outline.02.html). Math.grin.edu. Archived (https://web.archive.org/web/20100715042920/http://www.math.grin.edu/~rebelsky/Courses/CS302/99S/Outlines/outline.02.html) from the original on 15 July 2010. Retrieved 25 April 2010.
31. Eugene Loh (18 June 2010). "The Ideal HPC Programming Language" (http://queue.acm.org/detail.cfm?id=1820518). *Queue*. **8** (6). Archived (https://web.archive.org/web/20160304015345/http://queue.acm.org/detail.cfm?id=1820518) from the original on 4 March 2016.
32. "HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers" (http://www.netlib.org/benchmark/hpl). Archived (https://web.archive.org/web/20150215031500/http://www.netlib.org/benchmark/hpl/) from the original on 15 February 2015. Retrieved 21 February 2015.
33. Hopper (1978) p. 16.
34. Sammet (1969) p. 316
35. Sammet (1978) p. 204.
36. Richard L. Wexelblat: *History of Programming Languages*, Academic Press, 1981, chapter XIV.
37. François Labelle. "Programming Language Usage Graph" (http://www.cs.berkeley.edu/~flab/languages.html). *SourceForge*. Archived (https://web.archive.org/web/20060617055109/http://www.cs.berkeley.edu/%7Eflab/languages.html) from the original on 17 June 2006. Retrieved 21 June 2006.. This comparison analyzes trends in the 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.
38. Hayes, Brian (2006). "The Semicolon Wars". *American Scientist*. **94** (4): 299–303. doi:10.1511/2006.60.299 (https://doi.org/10.1511%2F2006.60.299).
39. Dijkstra, Edsger W. (March 1968). "Go To Statement Considered Harmful" (http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF) (PDF). *Communications of the ACM*. **11** (3): 147–148. doi:10.1145/362929.362947 (https://doi.org/10.1145%2F362929.362947). S2CID 17469809 (https://api.semanticscholar.org/CorpusID:17469809). Archived (https://web.archive.org/web/20140513014557/http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF) (PDF) from the original on 13 May 2014.

40. Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, Akihiko Nakase (December 1994). "KLIC: A Portable Implementation of KL1" *Proc. of FGCS '94, ICOT* Tokyo, December 1994. "Archived copy" (https://web.archive.org/web/20060925132105/http://www.icot.or.jp/ARCHIVE/HomePage-E.html). Archived from the original (http://www.icot.or.jp/ARCHIVE/HomePage-E.html) on 25 September 2006. Retrieved 9 October 2006. KLIC is a portable implementation of a concurrent logic programming language KL1.
41. Jim Bender (15 March 2004). "Mini-Bibliography on Modules for Functional Programming Languages" (http://readscheme.org/modules/). *ReadScheme.org*. Archived (https://web.archive.org/web/20060924085057/http://readscheme.org/modules/) from the original on 24 September 2006.
42. Michael Sipser (1996). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 978-0-534-94728-6. Section 2.2: Pushdown Automata, pp.101–114.
43. Jeffrey Kegler, "Perl and Undecidability (http://www.jeffreykegler.com/Home/perl-and-undecidability) Archived (https://web.archive.org/web/20090817183115/http://www.jeffreykegler.com/Home/perl-and-undecidability) 17 August 2009 at the Wayback Machine", *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.
44. Marty Hall, 1995, Lecture Notes: Macros (http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.html) Archived (https://web.archive.org/web/20130806054148/http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.html) 6 August 2013 at the Wayback Machine, PostScript version (http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.ps) Archived (https://web.archive.org/web/20000817211709/http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.ps) 17 August 2000 at the Wayback Machine
45. Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 18–19
46. Andrew Cooke. "Introduction To Computer Languages" (http://www.acooke.org/comp-lang.html). Archived (https://web.archive.org/web/20120815140215/http://www.acooke.org/comp-lang.html) from the original on 15 August 2012. Retrieved 13 July 2012.
47. 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) Archived (https://web.archive.org/web/20070129073839/http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00849.html) 29 January 2007 at the Wayback Machine (Alan Jeffrey, 17 December 2001) and Sound Generic Java type inference (http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00921.html) Archived (https://web.archive.org/web/20070129073849/http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00921.html) 29 January 2007 at the Wayback Machine (Martin Odersky, 15 January 2002). C#'s type system is similar to Java's, and uses a similar partial type inference scheme.
48. "Revised Report on the Algorithmic Language Scheme" (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-4.html). 20 February 1998. Archived (https://web.archive.org/web/20060714212928/http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-4.html) from the original on 14 July 2006.
49. Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism" (http://citeseer.ist.psu.edu/cardelli85understanding.html). *Manuscript (1985)*. Archived (https://web.archive.org/web/20060619072646/http://citeseer.ist.psu.edu/cardelli85understanding.html) from the original on 19 June 2006.
50. Steven R. Fischer, *A history of language*, Reaktion Books, 2003, ISBN 1-86189-080-X, p. 205

51. Éric Lévénez (2011). "Computer Languages History" (http://www.levenez.com/lang/). Archived (https://web.archive.org/web/20060107162045/http://www.levenez.com/lang/) from the original on 7 January 2006.
52. Jing Huang. "Artificial Language vs. Natural Language" (http://www.cs.cornell.edu/info/Projects/Nuprl/cs611/fall94notes/cn2/subsection3_1_3.html). Archived (https://web.archive.org/web/20090903084542/http://www.cs.cornell.edu/info/Projects/Nuprl/cs611/fall94notes/cn2/subsection3_1_3.html) from the original on 3 September 2009.
53. 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." ("PL/I" (http://www.encyclopediaofmath.org/index.php?title=PL/I&oldid=19175). *Encyclopedia of Mathematics*. Archived (https://web.archive.org/web/20120426010947/http://www.encyclopediaofmath.org/index.php?title=PL%2FI&oldid=19175) from the original on 26 April 2012. Retrieved 29 June 2006.). Ada and UNCOL had similar early goals.
54. Frederick P. Brooks, Jr.: *The Mythical Man-Month*, Addison-Wesley, 1982, pp. 93–94
55. Dijkstra, Edsger W. On the foolishness of "natural language programming." (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html) Archived (https://web.archive.org/web/20080120201526/http://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html) 20 January 2008 at the Wayback Machine EWD667.
56. Perlis, Alan (September 1982). "Epigrams on Programming" (http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html). *SIGPLAN Notices Vol. 17, No. 9*. pp. 7–13. Archived (https://web.archive.org/web/19990117034445/http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html) from the original on 17 January 1999.
57. Milner, R.; M. Tofte; R. Harper; D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press. ISBN 978-0-262-63181-5.
58. Kelsey, Richard; William Clinger; Jonathan Rees (February 1998). "Section 7.2 Formal semantics" (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%_sec_7.2). *Revised^{5} Report on the Algorithmic Language Scheme*. Archived (https://web.archive.org/web/20060706081110/http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%_sec_7.2) from the original on 6 July 2006.
59. ANSI – Programming Language Rexx, X3-274.1996
60. Steve, McConnell (2004). *Code complete* (https://archive.org/details/codecomplete0000mcco/page/590) (Second ed.). Redmond, Washington. pp. 590, 600 (https://archive.org/details/codecomplete0000mcco/page/590). ISBN 0735619670. OCLC 54974573 (https://www.worldcat.org/oclc/54974573).
61. See: Oracle America, Inc. v. Google, Inc.
62. "Guide to Programming Languages | ComputerScience.org" (https://www.computerscience.org/resources/computer-programming-languages/). *ComputerScience.org*. Retrieved 13 May 2018.
63. "The basics" (https://www.ibm.com/developerworks/library/os-erlang1/index.html). *ibm.com*. 10 May 2011. Retrieved 13 May 2018.
64. "HOPL: an interactive Roster of Programming Languages" (https://web.archive.org/web/20110220044217/http://hopl.murdoch.edu.au/). Australia: Murdoch University. Archived from the original (http://hopl.murdoch.edu.au/) on 20 February 2011. Retrieved 1 June 2009. "This site lists 8512 languages."

65. Mayer, Philip; Bauer, Alexander (2015). "An empirical analysis of the utilization of multiple programming languages in open source projects". *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering – EASE '15. New York, NY, USA: ACM. pp. 4:1–4:10. doi:10.1145/2745802.2745805 (https://doi.org/10.1145%2F2745802.2745805). ISBN 978-1-4503-3350-4. "Results: We found (a) a mean number of 5 languages per project with a clearly dominant main general-purpose language and 5 often-used DSL types, (b) a significant influence of the size, number of commits, and the main language on the number of languages as well as no significant influence of age and number of contributors, and (c) three language ecosystems grouped around XML, Shell/Make, and HTML/CSS. Conclusions: Multi-language programming seems to be common in open-source projects and is a factor which must be dealt with in tooling and when assessing development and maintenance of such software systems."
66. Abelson, Sussman, and Sussman. "Structure and Interpretation of Computer Programs" (https://web.archive.org/web/20090226050622/http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html). Archived from the original (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html) on 26 February 2009. Retrieved 3 March 2009.
67. Brown Vicki (1999). "Scripting Languages" (http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html). *mactech.com*. Archived (https://web.archive.org/web/20171202235828/http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html) from the original on 2 December 2017.
68. Georgina Swan (21 September 2009). "COBOL turns 50" (http://www.computerworld.com.au/article/319269/cobol_turns_50/). computerworld.com.au. Archived (https://web.archive.org/web/20131019181128/http://www.computerworld.com.au/article/319269/cobol_turns_50/) from the original on 19 October 2013. Retrieved 19 October 2013.
69. Ed Airey (3 May 2012). "7 Myths of COBOL Debunked" (http://www.developer.com/lang/other/7-myths-of-cobol-debunked.html). developer.com. Archived (https://web.archive.org/web/20131019171802/http://www.developer.com/lang/other/7-myths-of-cobol-debunked.html) from the original on 19 October 2013. Retrieved 19 October 2013.
70. Nicholas Enticknap. "SSL/Computer Weekly IT salary survey: finance boom drives IT job growth" (http://www.computerweekly.com/Articles/2007/09/11/226631/sslcomputer-weekly-it-salary-survey-finance-boom-drives-it-job.htm). *Computer Weekly*. Archived (https://web.archive.org/web/20111026035734/http://www.computerweekly.com/Articles/2007/09/11/226631/SSLComputer-Weekly-IT-salary-survey-finance-boom-drives-IT-job.htm) from the original on 26 October 2011. Retrieved 14 June 2013.
71. "Counting programming languages by book sales" (https://web.archive.org/web/20080517023127/http://radar.oreilly.com/archives/2006/08/programming_language_trends_1.html). Radar.oreilly.com. 2 August 2006. Archived from the original (http://radar.oreilly.com/archives/2006/08/programming_language_trends_1.html) on 17 May 2008.
72. 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
73. "Most Popular and Influential Programming Languages of 2018" (https://stackify.com/popular-programming-languages-2018/). stackify.com. 18 December 2017. Retrieved 29 August 2018.
74. Carl A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992, ISBN 0-262-57095-5, p. 1
75. "TUNES: Programming Languages" (http://tunes.org/wiki/programming_20languages.html). Archived (https://web.archive.org/web/20071020203251/http://tunes.org/wiki/programming_20languages.html) from the original on 20 October 2007.

76. Wirth, Niklaus (1993). "Recollections about the development of Pascal". *The second ACM SIGPLAN conference on History of programming languages – HOPL-II* (http://portal.acm.org/citation.cfm?id=155378). *Proc. 2nd ACM SIGPLAN Conference on History of Programming Languages*. Vol. 28. pp. 333–342. CiteSeerX 10.1.1.475.6989 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.475.6989). doi:10.1145/154766.155378 (https://doi.org/10.1145%2F154766.155378). ISBN 978-0-89791-570-0. S2CID 9783524 (https://api.semanticscholar.org/CorpusID:9783524).

Further reading

- Abelson, Harold; Sussman, Gerald Jay (1996). *Structure and Interpretation of Computer Programs* (https://web.archive.org/web/20180309173822/https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html) (2nd ed.). MIT Press. Archived from the original (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html) on 9 March 2018.
- Raphael Finkel: *Advanced Programming Language Design (https://web.archive.org/web/20141022141742/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 Victor 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