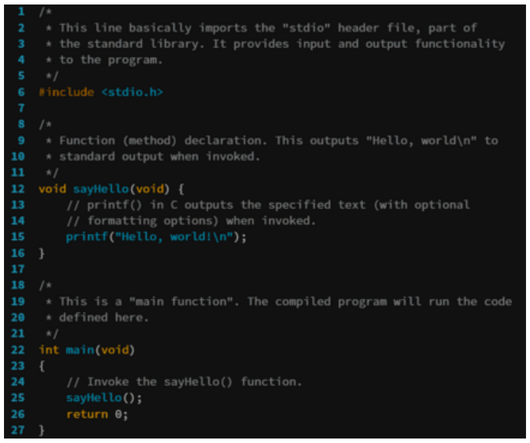
Wikipedia

Lenguaje de programación

Un **lenguaje de programación** es un sistema de notación para escribir programas de computadora . [1]



```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;
}
```

El código fuente de un programa informático en C. Las líneas grises son comentarios que explican el programa a los humanos. Cuando se compila y ejecuta , dará el resultado " ¡Hola, mundo! ".

Los lenguajes de programación se describen en términos de su sintaxis (forma) y semántica (significado), generalmente definidos por un lenguaje formal . Los lenguajes normalmente proporcionan características tales como un sistema de tipos , variables y mecanismos para el manejo de errores . Una implementación de un lenguaje de programación en forma de compilador o intérprete permite ejecutar programas , ya sea directamente o produciendo lo que en programación se conoce como ejecutable .

La arquitectura informática ha influido fuertemente en el diseño de los lenguajes de programación, y el tipo más común (lenguajes imperativos , que implementan operaciones en un orden específico) se desarrolló para funcionar bien en la popular arquitectura von Neumann . Si bien los primeros lenguajes de programación estaban estrechamente vinculados al hardware , con el tiempo han desarrollado una mayor abstracción para ocultar los detalles de implementación para una mayor simplicidad.

Se han desarrollado miles de lenguajes de programación, a menudo clasificados como imperativos, funcionales , lógicos u orientados a objetos , para una amplia variedad de usos. Muchos aspectos del diseño de lenguajes de programación implican compensaciones; por ejemplo, el manejo de excepciones simplifica el manejo de errores, pero a un costo de rendimiento. La teoría de los lenguajes de programación es el subcampo de la informática que estudia el diseño, implementación, análisis, caracterización y clasificación de los lenguajes de programación.

Definiciones

Hay una variedad de criterios que se pueden considerar al definir qué constituye un lenguaje de programación.

Lenguajes informáticos vs lenguajes de programación

The term *computer language* is sometimes used interchangeably with programming language.[2] 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.[3] 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.[4] One way of classifying computer languages is by the computations they are capable of expressing, as described by the theory of computation. The majority of practical programming languages are Turing complete,[5] and 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.[6][7] However, some authors restrict the term "programming language" to Turing complete languages.[1][8]

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.[9] 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.[10]

Domain and target

In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way.[11] 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.

The domain of the language is also worth consideration. 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]

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 supports adequate abstractions is expressed by the abstraction principle.[20] This principle is sometimes formulated as a recommendation to the programmer to make proper use of such abstractions.[21]

Historia

Early developments

The first programmable computers were invented at the end of the 1940s, and with them, the first programming languages.[22] The earliest computers were programmed in first-generation programming languages (1GLs), machine language (simple instructions that could be directly executed by the processor). This code was very difficult to debug and was not portable between different computer systems.[23] In order to improve the ease of programming, assembly languages (or second-generation programming languages—2GLs) were invented, diverging from the machine language to make programs easier to understand for humans, although they did not increase portability.[24]

Initially, hardware resources were scarce and expensive, while human resources were cheaper. Therefore, cumbersome languages that were time-consuming to use, but were closer to the hardware for higher efficiency were favored.[25] The introduction of high-level programming languages (third-generation programming languages—3GLs)—revolutionized programming. These languages abstracted away the details of the hardware, instead being designed to express algorithms that could be understood more easily by humans. For example, arithmetic expressions could now be written in symbolic notation and later translated into machine code that the hardware could execute.[24] In 1957, Fortran (FORmula TRANslation) was invented. Often considered the first compiled high-level programming language,[24][26] Fortran has remained in use into the twenty-first century.[27]

1960s and 1970s

Two people using an IBM 704 mainframe—the first hardware to support floating-point arithmetic—in 1957. Fortran was designed for this machine.[28][27]

Around 1960, the first mainframes—general purpose computers—were developed, although they could only be operated by professionals and the cost was extreme. The data and instructions were input by punch cards, meaning that no input could be added while the program was running. The languages developed at this time therefore are designed for minimal interaction.[29] After the invention of the microprocessor, computers in the 1970s became dramatically cheaper.[30] New computers also allowed more user interaction, which was supported by newer programming languages.[31]

Lisp, implemented in 1958, was the first functional programming language.[32] Unlike Fortran, it supports recursion and conditional expressions,[33] and it also introduced dynamic memory management on a heap and automatic garbage collection.[34] For the next decades, Lisp dominated artificial intelligence applications.[35] In 1978, another functional language, ML, introduced inferred types and polymorphic parameters.[31][36]

After ALGOL (ALGOrithmic Language) was released in 1958 and 1960,[37] it became the standard in computing literature for describing algorithms. Although its commercial success was limited, most popular imperative languages—including C, Pascal, Ada, C++, Java, and C#—are directly or indirectly descended from ALGOL 60.[38][27] Among its innovations adopted by later programming languages included greater portability and the first use of context-free, BNF grammar.[39] Simula, the first language to support object-oriented programming (including subtypes, dynamic dispatch, and inheritance), also descends from ALGOL and achieved commercial success.[40] C, another ALGOL descendant, has sustained popularity into the twenty-first century. C allows access to lower-level machine operations more than other contemporary languages. Its power and efficiency, generated in part with flexible pointer operations, comes at the cost of making it more difficult to write correct code.[31]

Prolog, designed in 1972, was the first logic programming language, communicating with a computer using formal logic notation.[41][42] With logic programming, the programmer specifies a desired result and allows the interpreter to decide how to achieve it.[43][42]

1980s to 2000s

A small selection of programming language textbooks

During the 1980s, the invention of the personal computer transformed the roles for which programming languages were used.[44] New languages introduced in the 1980s included C++, a superset of C that can compile C programs but also supports classes and inheritance.[45] Ada and other new languages introduced support for concurrency.[46] The Japanese government invested heavily into the so-called fifth-generation languages that added support for concurrency to logic programming constructs, but these languages were outperformed by other concurrency-supporting languages.[47][48]

Due to the rapid growth of the Internet and the World Wide Web in the 1990s, new programming languages were introduced to support Web pages and networking.[49] Java, based on C++ and designed for increased portability across systems and security, enjoyed large-scale success because these features are essential for many Internet applications.[50][51] Another development was that of dynamically typed scripting languages—Python, JavaScript, PHP, and Ruby—designed to quickly produce small programs that coordinate existing applications. Due to their integration with HTML, they have also been used for building web pages hosted on servers.[52][53]

2000s to present

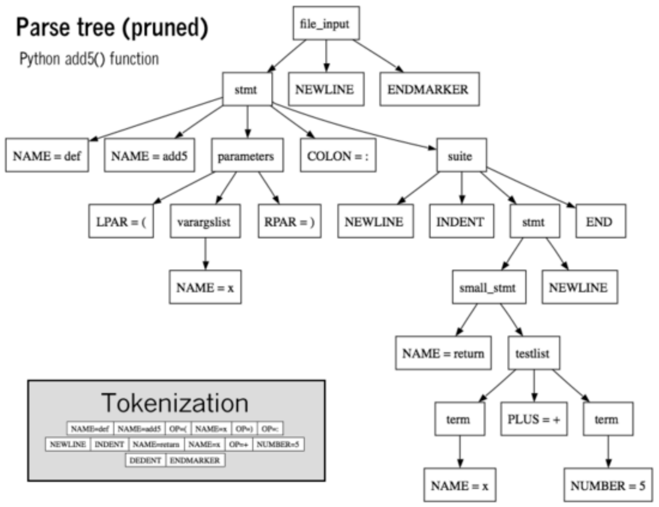
During the 2000s, there was a slowdown in the development of new programming languages that achieved widespread popularity.[54] One innovation was service-oriented programming, designed to exploit distributed systems whose components are connected by a network. Services are similar to objects in object-oriented programming, but run on a separate process.[55] C# and F# cross-pollinated ideas between imperative and functional programming.[56] After 2010, several new languages—Rust, Go, Swift, Zig and Carbon —competed for the performance-critical software for which C had historically been used.[57] Most of the new programming languages uses static typing while a few numbers of new languages use dynamic typing like Ring and Julia.[58][59]

Some of the new programming languages are classified as visual programming languages like Scratch, LabVIEW and PWCT. Also, some of these languages mix between textual and visual programming usage like Ballerina.[60][61][62][63] Also, this trend lead to developing projects that help in developing new VPLs like Blockly by Google.[64] Many game engines like Unreal and Unity added support for visual scripting too.[65][66]

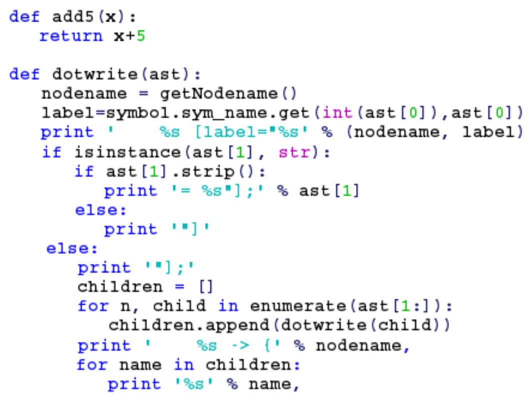
Elementos

Every programming language includes fundamental elements for describing data and the operations or transformations applied to them, such as adding two numbers or selecting an item from a collection. These elements are governed by syntactic and semantic rules that define their structure and meaning, respectively.

Syntax



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 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.

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, some programming languages 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.

The 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 the 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.[67] 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.[68] 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.[69]

Semantics

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

Static semantics

Static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms.[1] 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.[70] 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. Programming languages such as Java and C# have definite assignment analysis, a form of data flow analysis, as part of their respective 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 goes into formal semantics of programming languages, which allows 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.[71] 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 is 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.[71] 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.[71] 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 before a program executes, 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.[71]

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 functional languages such as Haskell and ML.[72] However, many manifestly-typed languages support partial type inference; for example, C++, Java, and C# all infer types in certain limited cases.[73] 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*.[71] 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, Ruby, Ring and Julia 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.[71] 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.[71] Strongly-typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl, Ring 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.[74][75][76]

It may seem odd to some professional programmers that C could be "weakly, statically typed". However, the use of the generic pointer, the **void*** pointer, does allow casting 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.

Concurrency

In computing, multiple instructions can be executed simultaneously. Many programming languages support instruction-level and subprogram-level concurrency.[77] By the twenty-first century, additional processing power on computers was increasingly coming from the use of additional processors, which requires programmers to design software that makes use of multiple processors simultaneously to achieve improved performance.[78] Interpreted languages such as Python and Ruby do not support the concurrent use of multiple processors.[79] Other programming languages do support managing data shared between different threads by controlling the order of execution of key instructions via the use of semaphores, controlling access to shared data via monitor, or enabling message passing between threads.[80]

Exception handling

Many programming languages include exception handlers, a section of code triggered by runtime errors that can deal with them in two main ways:[81]

- Termination: shutting down and handing over control to the operating system. This option is considered the simplest.
- Resumption: resuming the program near where the exception occurred. This can trigger a repeat of the exception, unless the exception handler is able to modify values to prevent the exception from reoccurring.

Some programming languages support dedicating a block of code to run regardless of whether an exception occurs before the code is reached; this is called finalization.[82]

There is a tradeoff between increased ability to handle exceptions and reduced performance.[83] For example, even though array index errors are common[84] C does not check them for performance reasons.[83] Although programmers can write code to catch user-defined exceptions, this can clutter a program. Standard libraries in some languages, such as C, use their return values to indicate an exception.[85] Some languages and their compilers have the option of turning on and off error handling capability, either temporarily or permanently.[86]

Diseño e implementación

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.[87][88] 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.[89] 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.[90] 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.[91]

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".[92] Alan Perlis was similarly dismissive of the idea.[93] 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[94] and Scheme[95] 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 formal language.

- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX[96]). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An implementation of a programming language is the conversion of a program into machine code that can be executed by the hardware. The machine code then can be executed with the help of the operating system.[97] The most common form of interpretation in production code is by a compiler, which translates the source code via an intermediate-level language into machine code, known as an executable. Once the program is compiled, it will run more quickly than with other implementation methods.[98] Some compilers are able to provide further optimization to reduce memory or computation usage when the executable runs, but increasing compilation time.[99]

Another implementation method is to run the program with an interpreter, which translates each line of software into machine code just before it executes. Although it can make debugging easier, the downside of interpretation is that it runs 10 to 100 times slower than a compiled executable.[100] Hybrid interpretation methods provide some of the benefits of compilation and some of the benefits of interpretation via partial compilation. One form this takes is just-in-time compilation, in which the software is compiled ahead of time into an intermediate language, and then into machine code immediately before execution.[101]

Idiomas propietarios

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,[102] 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.[103]

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 Ericsson's internal programming language.[104]

Open source programming languages are particularly helpful for open science applications, enhancing the capacity for replication and code sharing.[105]

Usar

Thousands of different programming languages have been created, mainly in the computing field.[106] Individual software projects commonly use five programming languages or more.[107]

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).[108] *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.[109]

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;[110][111] 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[112]
- the number of books sold that teach or describe the language[113]

- estimates of the number of existing lines of code written in the language – which may underestimate languages not often found in public searches[114]
- 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 (in descending order by overall popularity): Java, C, C++, Python, C#, JavaScript, VB .NET, R, PHP, and MATLAB.[115]

As of February 2024, the top five programming languages as measured by TIOBE index are Python, C, C++, Java and C#. TIOBE provide a list of top 100 programming languages according to popularity and update this list every month.[116]

Dialectos, sabores e implementaciones.

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 language has many dialects.

Clasificaciones

Programming languages are often placed into four main categories: imperative, functional, logic, and object oriented.[117]

- Imperative languages are designed to implement an algorithm in a specified order; they include visual programming languages such as .NET for generating graphical user interfaces. Scripting languages, which are partly or fully interpreted rather than compiled, are sometimes considered a separate category but meet the definition of imperative languages.[118]
- Functional programming languages work by successively applying functions to the given parameters. Although appreciated by many researchers for their simplicity and elegance,

problems with efficiency have prevented them from being widely adopted.[119]

- Logic languages are designed so that the software, rather than the programmer, decides what order in which the instructions are executed.[120]
- Object-oriented programming—whose characteristic features are data abstraction, inheritance, and dynamic dispatch—is supported by most popular imperative languages and some functional languages.[118]

Although markup languages are not programming languages, some have extensions that support limited programming. Additionally, there are special-purpose languages that are not easily compared to other programming languages.[121]

Ver también

- Comparison of programming

Computer programming

languages (basic instructions)

portal

- Comparison of programming languages
- Computer programming
- Computer science and Outline of computer science
- Domain-specific language
- Domain-specific modeling
- 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

- Reflective programming
- Scientific programming language
- Scripting language
- Software engineering and List of software engineering topics

Referencias

1. *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.*

2. *Robert A. Edmunds, The Prentice-Hall standard glossary of computer terminology, Prentice-Hall, 1985, p. 91*
3. *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*
4. *S.K. Bajpai, Introduction To Computers And C Programming, New Age International, 2007, ISBN 81-224-1379-X, p. 346*

5. *"Turing Completeness" (https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html) . www.cs.odu.edu. Archived (https://web.archive.org/web/20220816145137/https://cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html) from the original on 16 August 2022. Retrieved 5 October 2022.*

6. *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.*

7. *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."*

8. *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*.*

9. *R. Narasimhan, 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."*

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

11. *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."*

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

21. *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.*

22. *Gabbrielli & Martini 2023, p. 519.*

23. *Gabbrielli & Martini 2023, pp. 520–521.*

24. *Gabbrielli & Martini 2023, p. 521.*

25. *Gabbrielli & Martini 2023, p. 522.*

26. *Sebesta 2012, p. 42.*

27. *Gabbrielli & Martini 2023, p. 524.*

28. *Sebesta 2012, pp. 42–44.*

29. *Gabbrielli & Martini 2023, pp. 523–524.*

30. *Gabbrielli & Martini 2023, p. 527.*

31. *Gabbrielli & Martini 2023, p. 528.*

32. *"How Lisp Became God's Own Programming Language" (https://twobithistory.org/2018/10/14/lisp.html) . twobithistory.org. Archived (https://web.archive.org/web/20240410060444/https://twobithistory.org/2018/10/14/lisp.html) from the original on 10 April 2024. Retrieved 10 April 2024.*

33. *Sebesta 2012, pp. 47–48.*

34. *Gabbrielli & Martini 2023, p. 526.*

35. *Sebesta 2012, p. 50.*

36. *Sebesta 2012, pp. 701–703.*

37. *Gabbrielli & Martini 2023, pp. 524–525.*

38. *Sebesta 2012, pp. 56–57.*

39. *Gabbrielli & Martini 2023, p. 525.*

40. *Gabbrielli & Martini 2023, pp. 526–527.*

41. *Gabbrielli & Martini 2023, p. 531.*

42. *Sebesta 2012, p. 79.*

43. *Gabbrielli & Martini 2023, p. 530.*

44. *Gabbrielli & Martini 2023, pp. 532–533.*

45. *Gabbrielli & Martini 2023, p. 534.*

46. *Gabbrielli & Martini 2023, pp. 534–535.*

47. *Gabbrielli & Martini 2023, p. 535.*

48. *Sebesta 2012, p. 736.*

49. *Gabbrielli & Martini 2023, p. 536.*

50. *Gabbrielli & Martini 2023, pp. 536–537.*

51. *Sebesta 2012, pp. 91–92.*

52. *Gabbrielli & Martini 2023, pp. 538–539.*

53. *Sebesta 2012, pp. 97–99.*

54. *Gabbrielli & Martini 2023, p. 542.*

55. *Gabbrielli & Martini 2023, pp. 474–475, 477, 542.*

56. *Gabbrielli & Martini 2023, pp. 542–543.*

57. *Gabbrielli & Martini 2023, p. 544.*

58. *Bezanson, J., Karpinski, S., Shah, V.B. and Edelman, A., 2012. Julia: A fast dynamic language for technical computing. arXiv preprint arXiv:1209.5145.*

59. *Ayouni, M. and Ayouni, M., 2020. Data Types in Ring. Beginning Ring Programming: From Novice to Professional, pp.51-98.*

60. *Sáez-López, J.M., Román-González, M. and Vázquez-Cano, E., 2016. Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools. Computers & Education, 97, pp.129-141.*

61. *Fayed, M.S., Al-Qurishi, M., Alamri, A. and Al-Daraiseh, A.A., 2017, March. PWCT: visual language for IoT and cloud computing applications and systems. In Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing (pp. 1-5).*

62. *Kodosky, J., 2020. LabVIEW. Proceedings of the ACM on Programming Languages, 4(HOPL), pp.1-54.*

63. *Fernando, A. and Warusawithana, L., 2020. Beginning Ballerina Programming: From Novice to Professional. Apress.*

64. *Baluprithviraj, K.N., Bharathi, K.R., Chendhuran, S. and Lokeshwaran, P., 2021, March. Artificial intelligence based smart door with face mask detection. In 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS) (pp. 543-548). IEEE.*

65. *Sewell, B., 2015. Blueprints visual scripting for unreal engine. Packt Publishing Ltd.*

66. *Bertolini, L., 2018. Hands-On Game Development without Coding: Create 2D and 3D games with Visual Scripting in Unity. Packt Publishing Ltd.*

67. *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.*

68. *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.*

69. *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*

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

71. *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.*

72. *Leivant, Daniel (1983). Polymorphic type inference. ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Austin, Texas: ACM Press. pp. 88–98. doi:10.1145/567067.567077 (https://doi.org/10.1145%2F567067.567077) . ISBN 978-0-89791-090-3.*

73. *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.

74. *"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.*
75. *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.*

76. *Ayouni, M., 2020. Beginning Ring programming (Vol. 978, No. 1, pp. 4842-5832). Apress.*
77. *Sebesta 2012, p. 576.*
78. *Sebesta 2012, p. 579.*
79. *Sebesta 2012, p. 585.*
80. *Sebesta 2012, pp. 585–586.*
81. *Sebesta 2012, pp. 630, 634.*
82. *Sebesta 2012, p. 635.*
83. *Sebesta 2012, p. 631.*
84. *Sebesta 2012, p. 261.*
85. *Sebesta 2012, p. 632.*
86. *Sebesta 2012, pp. 631, 635–636.*
87. *Steven R. Fischer, A history of language, Reaktion Books, 2003, ISBN 1-86189-080-X, p. 205*

88. *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.*
89. *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.*

90. *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.*
91. *Frederick P. Brooks, Jr.: The Mythical Man-Month, Addison-Wesley, 1982, pp. 93–94*

92. 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.

93. 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.

94. Milner, R.; M. Tofte; R. Harper; D. MacQueen (1997). The Definition of Standard ML (Revised). MIT Press. ISBN 978-0-262-63181-5.

95. *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.*
96. *ANSI – Programming Language Rexx, X3-274.1996*
97. *Sebesta 2012, pp. 23–24.*
98. *Sebesta 2012, pp. 25–27.*
99. *Sebesta 2012, p. 27.*
100. *Sebesta 2012, p. 28.*
101. *Sebesta 2012, pp. 29–30.*
102. *See: Oracle America, Inc. v. Google, Inc.*

103. *"Guide to Programming Languages | ComputerScience.org" (https://www.computerscience.org/resources/computer-programming-languages/) . ComputerScience.org. Archived (https://web.archive.org/web/20180513223729/https://www.computerscience.org/resources/computer-programming-languages/) from the original on 13 May 2018. Retrieved 13 May 2018.*

104. *"The basics" (https://www.ibm.com/developerworks/library/os-erlang1/index.html) . ibm.com. 10 May 2011. Archived (https://web.archive.org/web/20180514064903/https://www.ibm.com/developerworks/library/os-erlang1/index.html) from the original on 14 May 2018. Retrieved 13 May 2018.*

105. *Abdelaziz, Abdullah I.; Hanson, Kent A.; Gaber, Charles E.; Lee, Todd A. (2023). "Optimizing large real-world data analysis with parquet files in R: A step-by-step tutorial" (https://doi.org/10.1002%2Fpds.5728) . Pharmacoepidemiology and Drug Safety. **33** (3): e5728. doi:10.1002/pds.5728 (https://doi.org/10.1002%2Fpds.5728) . PMID 37984998 (https://pubmed.ncbi.nlm.nih.gov/37984998) .*

106. *"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."*

107. *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, US: 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 that must be dealt with in tooling and when assessing the development and maintenance of such software systems."

108. *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.*

109. *Vicki, Brown; Morin, Rich (1999). "Scripting Languages" (http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html) . MacTech. 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.*

110. *Georgina Swan (21 September 2009). "COBOL turns 50" (https://web.archive.org/web/20131019181128/http://www.computerworld.com.au/article/319269/cobol_turns_50/) . Computerworld. Archived from the original (http://www.computerworld.com.au/article/319269/cobol_turns_50/) on 19 October 2013. Retrieved 19 October 2013.*

111. *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.*

112. *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.*

113. *"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.*

114. *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*

115. *"Most Popular and Influential Programming Languages of 2018" (https://stackify.com/popular-programming-languages-2018/) . stackify.com. 18 December 2017. Archived (https://web.archive.org/web/20180830004924/https://stackify.com/popular-programming-languages-2018/) from the original on 30 August 2018. Retrieved 29 August 2018.*

116. *"TIOBE Index" (https://www.tiobe.com/tiobe-index/) . Archived (https://web.archive.org/web/20231015080735/https://www.tiobe.com/tiobe-index/) from the original on 15 October 2023. Retrieved 24 February 2024.*

117. *Sebesta 2012, p. 21.*

118. *Sebesta 2012, pp. 21–22.*

119. *Sebesta 2012, p. 12.*

120. *Sebesta 2012, p. 22.*

121. *Sebesta 2012, pp. 22–23.*

Otras lecturas

- 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.
- 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/) Archived (https://web.archive.org/web/20210430210417/http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/) 30 April 2021 at the Wayback Machine.
- Gabbrielli, Maurizio; Martini, Simone (2023). *Programming Languages: Principles and Paradigms* (2nd ed.). Springer. ISBN 978-3-031-34144-1.
- 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.
- Sebesta, Robert W. (2012). *Concepts of Programming Languages* (10 ed.). Addison-Wesley. ISBN 978-0-13-139531-2.

- 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.
- Wilson, Leslie B. (2001). *Comparative Programming Languages, Third Edition*. Addison-Wesley. ISBN 0-201-71012-9.

rogramming language at Wikipedia's sister projects:

- **Definitior** from Wiktionar
- **Media** frc Common
- **Quotatior** from Wikiquote
- **Textbook** from Wikibook
- **Resource** from Wikiversit
- **Data** fron Wikidata