A Book Chapter

By Narendra Kumar Chahar

on Programming Languages

Index:

Chapter 1

Programming Languages

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

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard loomsand player pianos[citation needed]. Thousands of different programming languages have been created, mainly in the computer field, with many being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). Some languages are defined by a specification document (for example, the C programming language is specified by an ISO Standard), while other languages, such as Perl 5 and earlier, have a dominant implementation that is used as a reference.

History of programming languages:

The first programming languages predate the modern computer. At first, the languages were codes.

The Jacquard loom, invented in 1801, used holes in punched cards to represent sewing loom arm movements in order to generate decorative patterns automatically.

During a nine-month period in 1842-1843, Ada Lovelace translated the memoir of Italian mathematician Luigi Menabrea about Charles Babbage's newest proposed machine, theAnalytical Engine. With the article, she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the Engine, recognized by some historians as the world's first computer program.[1]

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

The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers.

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

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

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

The 1940s

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

In 1948, Konrad Zuse published a paper about his programming language Plankalkül. However, it was not implemented in his lifetime and his original contributions were isolated from other developments.

Some important languages that were developed in this period include:

- 1943 - Plankalkül (Konrad Zuse), designed, but unimplemented for a half-century
- 1943 - ENIAC, Electric Numerical Integrator And Computer, machine-specific codeset appearing in 1948.[2]
- 1949 - 1954 — a series of machine-specific mnemonic instruction sets, like ENIAC's, beginning in 1949 with C-10 for BINAC (which later evolved into UNIVAC).[3] Each codeset, or instruction set, was tailored to a specific manufacturer.

[edit]The 1950s and 1960s

In the 1950s, the first three modern programming languages whose descendants are

still in widespread use today were designed:

- FORTRAN (1955), the "FORmula TRANslator", invented by John Backus et al.;
- LISP (1958), the "LISt Processor", invented by John McCarthy et al.;
- COBOL,(1959) the COmmon Business Oriented Language, created by the Short-Range Committee, heavily influenced by Grace Hopper.

Another milestone in the late 1950s was the publication, by a committee of American and European computer scientists, of "a new language for algorithms"; the ALGOL 60 Report (the "ALGOrithmic Language"). This report consolidated many ideas circulating at the time and featured two key language innovations:

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

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

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

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

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

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

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

Some important languages that were developed in this period include:

- 1951 - Regional Assembly Language
- 1952 - Autocode
- 1954 - IPL (forerunner to LISP)
- 1955 - FLOW-MATIC (forerunner to COBOL)
- 1957 - FORTRAN (First compiler)
- 1957 - COMTRAN (forerunner to COBOL)

- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (forerunner to COBOL)
- 1959 - COBOL
- 1959 - RPG
- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (forerunner to C)
- 1964 - BASIC
- 1964 - PL/I
- 1967 - BCPL (forerunner to C)

1968-1979: establishing fundamental paradigms

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

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

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

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

Some important languages that were developed in this period include:

- 1968 - Logo
- 1969 - B (forerunner to C)
- 1970 - Pascal
- 1970 - Forth

- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1975 - Scheme
- 1978 - SQL (initially only a query language, later extended with programming constructs)

The 1980s: consolidation, modules, performance

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

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

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

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

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

Some important languages that were developed in this period include:

- 1980 - C++ (as C with classes, name changed in July 1983)
- 1983 - Ada
- 1984 - Common Lisp
- 1984 - MATLAB
- 1985 - Eiffel
- 1986 - Objective-C

- 1986 - Erlang
- 1987 - Perl
- 1988 - Tcl
- 1988 - Mathematica
- 1989 - FL (Backus);

The 1990s: the Internet age

The rapid growth of the Internet in the mid-1990s was the next major historic event in programming languages. By opening up a radically new platform for computer systems, the Internet created an opportunity for new languages to be adopted. In particular, the Java programming language rose to popularity because of its early integration with the Netscape Navigator web browser, and various scripting languages achieved widespread use in developing customized application for web servers. The 1990s saw no fundamental novelty inimperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many "rapid application development" (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention. More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain.[citation needed] Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web.

Some important languages that were developed in this period include:

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

Features of programming Languages:

• The vocabulary of the language should resemble English (or some other human language). Symbols, abbreviations, and jargon should be avoided unless they're already familiar to most people.

• Programs should consist mostly of instructions; tedious declarations should be kept to a minimum.

• The language and its class or function library should be fully documented. Source code, even if provided, is no substitute for documentation. At least for beginners and part-timers, a page of documentation is far more intelligible than a page of source code. Any function will generally invoke a number of other functions, and therefore the source code is completely unintelligible unless you already know what every function does.

• There should be no need to manipulate pointers explicitly, and no means of doing so. Pointers are tedious to deal with and they're a fruitful source of bugs; they should be managed by the language and not by the programmer. This approach has been tried and shown to be feasible in a number of languages.

• The language should provide arrays of unlimited size: there should be no need to declare array bounds. Sorting facilities should be included as standard; we shouldn't have to write our own sort routines.

• Integers of unlimited size (as in Smalltalk) are nice to have in principle, though in practice not urgently needed for most programs.

• The language should provide full facilities for handling a graphical user interface. These should be defined as a standard part of the language, irrespective of the operating system in use. Algol-60 was a failure in practice because it failed to define input/output as a part of the language, and therefore its input/output statements varied from one implementation to another. Input/output now includes the graphical user interface, and any programming language should take responsibility for it.

• The language should probably be object-oriented. However, I've never actually written a pure object-oriented program, so I say this with more faith than experience.

• In Smalltalk, everything you write yourself is included in the "image file" and effectively becomes part of the language from then on. I dislike this approach: I prefer to keep one project separate from another and my projects separate from the libraries supplied with the language. When I create classes that I want to use in more than one project, I should be able to put them in a Personal Library folder or something.

• Any concept that can't easily be explained to children probably shouldn't be included in the language. Part-time programmers don't want to struggle with difficult concepts, they just want to get a job done quickly and easily.

Programming Paradigms:

There are four basic computational models that describe most programming today, Imperative, applicative, rule based and object oriented.

Imperative: The language provides statements, such as assignment statements, which explicitly change the state of the memory of the computer.

Functional: In this paradigm we express computations as the evaluation of mathematical functions.

Logic: In this paradigm we express computation in exclusively in terms of mathematical logic

Object-Oriented: In this paradigm we associate behaviour with data-structures called "objects" which belong to classes which are usually structured into a hierarchy.

The paradigms are not exclusive, but reflect the different emphasis of language designers. Most practical imperative, functional and object-oriented languages embody features of more than one paradigm.

The Functional Paradigm

In this we emphasise the idea of computation as being about evaluating mathematical functions combined in expressions. While all languages try to provide a representation of basic functions like addition, functional languages support a functional way of expressing computations on large and complex structures, although some such as Scheme also have imperative features. In a pure functional language, a mathematical identity like:

fred(x) + fred(x) = 2*fred(x)

should hold. This is not necessarily the case for a non-functional language, for example in Pascal or C fred might be a procedure which had a side-effect, so that calling it twice has a different effect from calling it once. For example fred might contain the assignment g := g+1 where g is a global variable. The primary motivation of writing functionally is to develop programs whose behaviour can easily be analysed mathematically, and in any case is easy to predict and understand.

The same non-functional aspect holds also for Java. A method call fred(x) will commonly have a side-effect.

However it has been difficult to design languages under the functional paradigm which produce programs which run as fast as under the imperative paradigm. With the high performance of modern computers, this matters less for many applications

than the ability to write correct programs. The functional paradigm is hard to implement efficiently because if a storage location is once used to hold a value it is not obvious when it can be re-used - a computer running a program using the functional paradigm can spend a lot of effort determining the reusability of store.

Another way to think of the functional paradigm is to regard it as a way of taking to its limit the advice to avoid harmful side-effects in a program.

The Imperative Paradigm

Languages which reflect this paradigm recognise the fact computers have re-usable memory that can change state. So they are characterised by statements, which affect the state of the machine, for example.

x := x+1

This can only be understood mathematically by associating a sequence of values with x let us say x_1, x_2,..., where x_t denotes the value the variable x has at some time t. Thus the above statement can be translated into mathematics as

$x_{t+1} = x_t + 1$

This kind of reasoning is discussed in CS250. It gets increasingly hard to do as the state-changes get ever more complex (e.g. by assigning to data-structure components). However imperative languages can relatively easily be translated into efficient machine-code, and so are usually considered to be highly efficient. Many people also find the imperative paradigm to be a more natural way of expressing themselves.

Languages which use the imperative paradigm commonly have functional features - for example the basic functions of arithmetic (addition, subtraction...) are provided.

The Logic Paradigm(Rule based)

While the functional paradigm emphasises the idea of a mathematical function, the logic paradigm focusses on predicate logic, in which the basic concept is a relation. Logic languages are useful for expressing problems where it is not obvious what the functions should be. Thus, for example where people are concerned, it is natural to use relations.

For example consider the uncle relationship: a given person can have many uncles, and a another person can be uncle to many nieces and nephews.

Let us consider now how we can define the brother relation in terms of simpler relations and properties father, mother, male. Using the Prolog logic language one can say:

```
brother(X,Y)      /* X is the brother of Y                  */
    :-            /* if there are two people F and M for which*/
    father(F,X),  /*          F is the father of X          */
    father(F,Y),  /*    and   F is the father of Y          */
    mother(M,X),  /*    and   M is the mother of X          */
    mother(M,Y),  /*    and   M is the mother of Y          */
    male(X).      /*    and   X is male                     */
```

That is X is the brother of Y if they have the same father and mother and X is male. Here ":-" stands for logical implication (written right to left).

Mathematical logic has always had an important role in computation, since boolean logic is the basis of the design of the logic circuits which form the basis of any computer. In the logic paradigm we make use of a more advanced construct, namely predicate logic, to give us languages of enhanced expressive power.

The Object-Oriented Paradigm

The Object-Oriented paradigm (often written O-O) focusses on the objects that a program is representing, and on allowing them to exhibit "behaviour". This is contrasted with the typical approach in the imperative paradigm, in which one typically thinks of operating on data with procedures. In the imperative paradigm typically the data are passive, the procedures are active. In the O-O paradigm, data is combined with procedures to give objects, which are thereby rendered active. For example, in the imperative paradigm, one would write a procedure which prints the various kinds of object in the program. In the O-O paradigm, each object has a print-method, and you "tell" an object to print itself.

It is however possible to use certain non-object-oriented languages to write object-oriented programs. What is required is the ability to create data-structures that contain machine code, or pointers to machine code. This is possible in the C language and in most functional languages (where functions are represented as code).

Objects belong to classes. Typically, all the objects in a given class will have the same kinds of behaviour.

Classes are usually arranged in some kind of class hierarchy. This hierarchy can be thought of as representing a "kind of" relation. For example, a computational model of the University might need a class person to represent the various people who make up the University. A sub-class of person might be a student; students are a kind of person. Another sub-class might be professor. Both students and professors can

exhibit the same kinds of behaviour, since they are both people. They both eat drink and sleep, for example. But there are kinds of behaviour that are distinctive: professors pontificate for example.

References:

[1] Wirth, Niklaus. "The programming language Pascal." *Acta informatica* 1.1 (1971): 35-63.
[2] Iverson, Kenneth E. "A programming language." *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962.
[3] Wirth, Niklaus. "The programming language Oberon." *Softw., Pract. Exper.* 18.7 (1988): 671-690.
[4] Iverson, Kenneth E. "A programming language." *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962.
[5] Hansen, Per Brinch. "The programming language concurrent pascal." *IEEE Transactions on Software Engineering* 2 (1975): 199-207.
[6] Ghezzi, Carlo, and Mehdi Jazayeri. *Programming language concepts*. John Wiley & Sons, 2008.