MANCHESTER
1824

The University
of Manchester

Programming Languages: An Applied Mathematics View

Higham, Nicholas J.

2015

MIMS EPrint: **2015.89**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Programming Languages: An Applied Mathematics View†

Nicholas J. Higham

The purpose of this article is to give an overview of computer programming languages from the point of view of applied mathematics. The historical development is emphasized because modern languages have been strongly influenced by those that came before and indeed the oldest language of all, Fortran, is still widely used. Figure 2 shows the major relationships between the languages discussed in this article.

1 The Early Days

The first digital stored-program computers were programmed by directly entering the low-level instructions, represented by binary numbers, that the central processing unit understood; see figure 1. This was tedious and error-prone, so assembly languages were developed that allowed the instructions to be entered as mnemonics, which were then translated by software (the assembler) into the corresponding binary instructions. To add 5 to a number stored in a memory location one might write a sequence of assembly language instructions such as `LDA P` (load the contents of memory location `P` into the accumulator), `ADC #5` (add 5 to the accumulator), `STA Q` (store the contents of the accumulator in memory location `Q`). Assembly language requires the programmer to work at the level of individual machine instructions and is far removed from mathematical notation.

It was a major step forward when John Backus and his colleagues at IBM designed the language *Fortran* and in 1957 distributed a Fortran compiler for the IBM 704 computer. A *compiler* translates a program written in a high-level language into a sequence of machine instructions that can then be directly executed. Standing for "formula translation", Fortran allowed mathematical expressions to be expressed in a natural algebraic notation, such as `Q = P + 5` for the example above. It also included many of the features we take for granted in programming languages today, such as loops, conditional tests, arrays, and elementary functions. Fortran was a huge success and it became the first programming language to be standardized, as American National Standards Institute (ANSI) standard Fortran 66 (where the digits denote the year of adoption of the standard). Standardization was important for expanding the number of compiler implementations of the language and aiding the portability of programs from one system to another. Fortran 66 included subroutines and functions, and also supported three floating-point data types: real, double precision, and complex.

Figure 1 The first program for a stored-program computer: a version of Tom Kilburn's highest factor routine that first ran on the Manchester "Baby" on June 21, 1948. Taken from G. C. Tootill's notebook. Copyright of The University of Manchester.

Subroutines and functions are examples of *subprograms*: sequences of code forming essentially separate programs that can be called with different input arguments from a main program or other subprogram. They are essential in mathematical computation for encapsulating basic operations such as adding two vectors or finding a norm of a vector, as well as for higher level tasks such as finding the roots of a polynomial or solving a differential equation. Arguments to a subprogram can be passed in at least two ways. In *call by value* the argument is evaluated at the time of the subprogram call and its value is copied into the formal param-

†. Author's final version, before copy editing and cross-referencing, of: N. J. Higham. Programming languages: An applied mathematics view. In N. J. Higham, M. R. Dennis, P. Glendinning, P. A. Martin, F. Santosa, and J. Tanner, editors, *The Princeton Companion to Applied Mathematics*, pages 828–839. Princeton University Press, Princeton, NJ, USA, 2015.

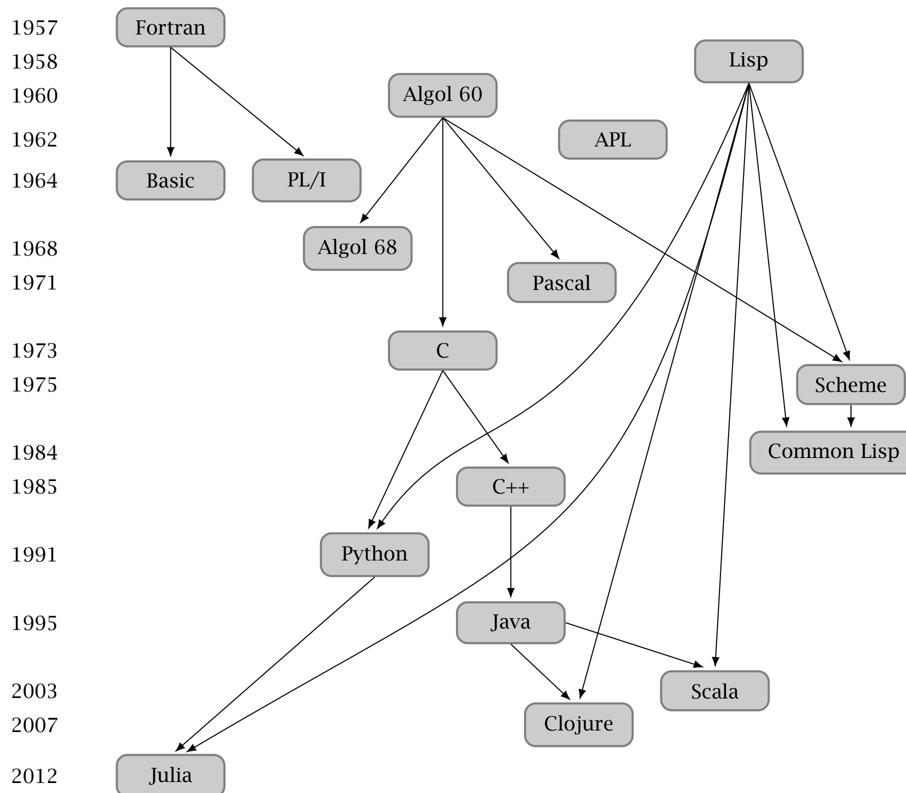

Figure 2 Timeline of selected programming languages, with major influences denoted by arrows.

eter inside the subprogram. In *call by reference* the address of the parameter is passed, so that the actual and formal parameters effectively share the same memory locations. An important difference between call by value and call by reference is that in the latter case any change made to the argument within the subprogram also changes the actual argument. In Fortran all parameters are passed by reference.

The first commercial Fortran textbook was "A Guide to Fortran Programming" by Daniel McCracken, published in 1961. The author has stated that only a couple of programs in the book had been tested because machine time cost $46 per hour!

Lisp, invented by John McCarthy in 1958, is the second oldest language still in wide use. The name stands for "list processor" and, as the name suggests, Lisp is based on list data structures. Lisp is well suited to *functional programming*, in which programs are entirely expressed in terms of mathematical functions (in particular, function application is the only control structure) and functions have no "side-effects", that is, they do not do anything except return a value. Lisp programs look completely different to those written in an *imperative language* such as Fortran, not least due to the profusion of parentheses and the use of prefix notation (the sum `1 + 2 + 3` is expressed as `(+ 1 2 3)`—see section 5.4). While Lisp is rarely used for floating-point computation it is well suited to symbolic computation, and it is the language in which the popular Emacs editor is mostly written. Lisp also has intrinsic mathematical interest due to its close relation to Alonzo Church's lambda calculus. The "if-then-else" construct first appeared in Lisp. Lisp has various dialects, including Scheme (used particularly for teaching) and Common Lisp.

Fortran had been developed in an ad hoc manner. A different language called *Algol 60* was produced in 1960 through the efforts of an international committee. The language was described in a formal notation, later called Backus–Naur form. Algol 60 was based on nested blocks delimited by `begin` and `end` statements with the scope of a variable (the region of the program in

which it is valid) restricted to the enclosing block, and it allowed for dynamic arrays, whose size is determined during execution of the program. It became the "official" language for publishing mathematical software in the 1960s (notably for the first six years of the journal *Communications of the ACM*, which began in 1960), and a strong competitor to Fortran for practical use. However the language ultimately did not succeed, for a variety of reasons, including the fact that it did not define any input–output facilities (making it impossible to write a portable "Hello, world!" program). Nevertheless, some influential early mathematical software was published in Algol 60, notably linear algebra software in the journal *Numerische Mathematik*, later collected into a 1971 volume of the *Handbook for Automatic Computation* series. In late 2014 it was reported that a language called JOVIAL based on a 1958 version of Algol was still in use in the UK air traffic control system.

Algol 60 greatly influenced future languages, such as *Algol 68* (1968), a more rigorously defined language designed by a working group of the International Federation for Information Processing, which was mainly of interest to computer science researchers, and *Pascal*, published in 1971 by Niklaus Wirth, which is a much smaller and simpler language than Algol 68. Pascal was widely taught in universities through the 1980s, as it promoted the notion of structured programming (see section 5.17) and so provided a way to avoid the hard to read "spaghetti code" that could easily be produced in Fortran 66. It also achieved wide use in industry, thanks to the availability of compilers on early PCs and Macintosh computers. However, Pascal was not well-suited to numerical programming, not least due to its support for only one type of floating-point variable (real) and the absence of an exponentiation operator.

An influential early textbook was George Forsythe and Cleve Moler's *Computer Solution of Algebraic Equations*, published in 1967. It contained listings of programs written in Algol 60, Fortran, and PL/I for solving a square linear system of equations $Ax = b$. (*PL/I* (1964) was a large language that was not widely adopted for scientific computing; Edsger Dijkstra said that "Using PL/I must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit.") These codes were part of a long sequence that led to the Fortran linear system solver in the LINPACK (1979) library.

Basic was invented at Dartmouth College in 1964 by John Kemeny and Thomas Kurtz in order to teach programming to students who did not necessarily have a science background. At Dartmouth, Basic was used on a time-sharing system, which allowed the programmer to interact with the computer via a terminal, as opposed to the usual batch processing of the time in which jobs were prepared on punched cards and handled by computer operators. The original Basic was in some respects a simplified Fortran, with only one data type (double precision), free-form input, the keyword `LET` required before every assignment, numbered lines, and a `GOTO` command whose destination was a line number. Many early personal computers, including the IBM PC, provided versions of Basic (usually based on Microsoft Basic), typically built into the firmware of the machine. *Visual Basic*, introduced by Microsoft in 1991, included features to aid in the development of graphical user interface (GUI) applications and it continues to exist as part of the .NET framework. Although a language often associated with writing games (such as the classic Star Trek game originating on 1970s minicomputers), Basic was a capable language for numerical computations and its accessibility on microcomputers led to it being widely used in mathematical research and teaching, including by this author. Basic was often implemented with an *interpreter*, which translates and executes each statement in the source code before going on to the next statement.

Another 1960s development was the language *APL*, implemented at IBM in 1965. It takes its name, and much of its notation, from the 1962 book *A Programming Language* by Kenneth Iverson. It is unusual in using non-ASCII characters to represent operators and functions, which make possible very concise programs that are often criticized as being cryptic. The notation $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ for the floor and ceiling functions originates in Iverson's book and is used (as functions ⌊ and ⌈) in APL. Indeed, APL has been described as an "executable notation." APL has powerful array processing and is normally interpreted. It was never widely adopted but has been influential and is still in use today.

2 The Modern Era

The language *C* (1973), by Dennis Ritchie, is a compact language in which the Unix operating system was mainly implemented. C has float and long floating-point data types, corresponding to single and double precision, respectively. Arguments to C functions are passed by value, but a pointer can be passed in order to achieve call by reference. The syntax is terse and powerful. C has been remarkably successful, for

several reasons. First, its operations and types map directly to the hardware, making it easy to write programs that carry out low-level system tasks and making it possible for compilers to produce very efficient code. Second, an ANSI/ISO standard was produced in 1989 (and revised in 1999 and 2011), aiding portability of programs. Third, C has remained more free from proprietary extensions than other languages.

The language *C++* by Bjarne Stroustrup (1985) is a descendant of C that is a superset, but for minor details, and adds better type checking, flexible data abstraction mechanisms, and support for object-oriented programming. *Data abstraction* allows the programmer to specify user-defined types, called *classes* in C++, and isolates how they are represented from how they are used—in other words, it hides the implementation details within the implementation of the types. *Object-oriented programming* is a methodology based on a hierarchy of classes and *objects*, which are specific instances of the classes with their own characteristics. One of the most popular uses of object-oriented programming is in developing GUIs. C++ also supports "generic programming", through the use of templates, whereby code can be written with parametrized types.

Throughout the history of computing, new programming languages have regularly been designed, with various goals, including providing a better general-purpose language or providing a language tailored to specific purposes, such as system programming tasks.

Java, developed by James Gosling at Sun Microsystems in 1995, is a widely-used object-oriented language with a syntax similar to that of C++. It compiles to a machine-independent bytecode that runs in the Java virtual machine (JVM), and a JVM is provided for each machine on which Java is to be used. The initial version of Java required bitwise reproducibility of floating-point arithmetic across different machines. While superficially an attractive feature it inhibited common compiler optimizations as well as the use of extended precision registers. These over-restrictive floating-point semantics were relaxed in later versions of Java, but other aspects such as the lack of complex arithmetic continue to hinder its use for numerical computation. JVMs exploit *just-in-time compilation*, in which Java bytecode is compiled into native machine code at run time. The JVM has importance beyond Java: some more recent languages such as *Scala* (2003) and *Clojure* (a dialect of Lisp created in 2007) compile to JVM bytecode.

Of the many languages introduced since C++, the most important from the computational mathematics point of view is *Python* (1991), designed by Guido van Rossum. Python is a *dynamic language*, which means that it lies somewhere between an interpreted language and a compiled language, with many features of the latter. It supports several programming paradigms, including object orientation and functional programming. Its success in scientific computing stems to a large extent from its libraries, which provide core computational and graphics capabilities (NumPy, SciPy, and matplotlib), and from its ability to integrate components written in other languages, such as C and Fortran. It has been said that "one doesn't need to switch to Python, only to know where to use it." Python was designed to be a readable language and its expression syntax is similar to that of C.

The newest language discussed here is *Julia* (2012), designed specifically for high-performance scientific computing. Julia is a dynamic language that achieves speed approaching that of compiled C code, in part due to just-in-time compilation using the LLVM compiler infrastructure. A distinctive feature of Julia is its exploitation of *multiple dispatch*, which allows a function to exist in several forms operating on different data types, with the appropriate version being called at run time based on the actual arguments supplied. An interesting feature of Julia is that it allows the user to view the underlying assembly language that the language generates. Viewing these low-level operations can provide much insight into how the language works and its efficiency; see figure 3.

The Fortran standard has undergone regular revisions, known as "Fortran xy", where xy is 77, 90, 95, 2003, or 2008 and is related to the year of publication of the standard. Fortran 77 introduced an if-then-else construct, improved input/output, and a character data type. Fortran 90 incorporated dynamic array allocation, operations on arrays, modules (a mechanism for packaging data, derived types, subprograms, and interface blocks), recursive subprograms, numeric inquiry functions, and parametrized intrinsic types. Later revisions have introduced support for object-oriented programming and for handling exceptions in IEEE floating-point arithmetic, and interoperability with C.

3 Parallelism

Most of the languages mentioned above do not include facilities for managing execution of codes in parallel,

```
In [1]: f(x,y) = x*y
Out[1]: f (generic function with 1 method)
In [2]: @code_native f(3,5)
.text
Filename: In[1]
Source line: 1
       push RBP
       mov  RBP, RSP
Source line: 1
       imul RDI, RSI
       mov  RAX, RDI
       pop  RBP
       ret
```

```
In [3]: @code_native f(3.0,5.0)
.text
Filename: In[1]
Source line: 1
       push RBP
       mov  RBP, RSP
Source line: 1
       mulsd XMM0, XMM1
       pop   RBP
       ret
```

Figure 3 A short Julia session, run from within a Jupyter notebook. The text that follows "`In [·]:`" on a line is user input. The definition of the function `f` does not specify the types of the arguments. Julia generates different x86 assembler code depending on whether the actual arguments are integers (as in `In [2]`) or floating-point numbers (as in `In [3]`).

that is, for specifying how a computation is to be broken up and executed by different processors simultaneously. Various extensions of existing languages have been proposed for parallel computing, but generally they have not achieved widespread or long-term use. Two widely used systems for parallel computing are the *Message Passing Interface* (MPI) for distributed memory systems and Open MP for shared memory systems. Both are implemented as application programming interfaces (APIs) that can be invoked from languages such as C, C++, and Fortran. For expressing parallelism on specialist devices such as graphics processing units (GPUs), specialist languages are available, such as *CUDA* for NVIDIA GPUs and *Open Computing Language* (OpenCL) for GPUs and heterogeneous platforms in general.

4 Problem Solving Environments

Nowadays a large part of scientific computing is done within environments that provide a programming language, an interactive command window with the display of graphics, and the ability to export graphics and more generally publish documents to HTML, PDF, TeX, and so on. They usually also have the ability to mix numerical and symbolic computing and by default display the result of assignments in the command window. The term *problem solving environments* (PSEs) is used for such systems, of which there are many.

PSEs have dynamic languages that, combined with the interactive interface, avoid the edit-compile-run cycle of languages such as C and Fortran. They allow quick coding without the need to define the types of variables before use. Moreover, a PSE language typically includes high level constructs that would correspond in a traditional language to many lines of code, such as a command to find the indices of the largest element(s) of an array or to compute the eigensystem of a matrix. Since it is generally accepted that a programmer's productivity, measured in the number lines of code written, is independent of the language, it follows that using a higher level language should allow the programmer to achieve more in a given time. On the other hand, PSEs usually do not execute code as fast as a compiled language.

The oldest PSE is *MATLAB*, originally written in Fortran in 1978 by Cleve Moler as a means of providing students with easy access to the EISPACK and LINPACK linear algebra program libraries. Rewritten in C, MATLAB was released as a commercial product in 1984 by The MathWorks. The fundamental data type in MATLAB is a matrix and MATLAB fully supports complex arithmetic.

An interesting feature of MATLAB is that much of it is written in MATLAB, in the form of M-files containing MATLAB commands. Certain key functions are written in C or call vendor-supplied Basic Linear Algebra Subprograms (BLAS) or LAPACK codes. MATLAB programs tend to be much shorter than their equivalents in compiled languages and yet, depending on the nature of the code, they can run at similar speed. Because of the ease and economy of coding, and the interactive interface which aids debugging, MATLAB is often used as a *prototyping tool*: an environment for developing and testing ideas before implementing them in a language such as C or Fortran.

GNU Octave is free software with many of the features of MATLAB and a largely compatible syntax, so that carefully coded programs can run in both MATLAB and Octave. *Scilab* is another open source alternative to MATLAB, but it is less compatible with MATLAB than Octave.

Maple started out as a computer algebra system developed at the University of Waterloo in 1980. It is now a commercial product sold by Waterloo Maple and has all the usual features of a PSE.

Mathematica, by Wolfram Research Software, had a notebook interface from its first release in 1988, showing program code, output with typeset mathematics, and graphics in a single window. It supports procedural, functional, rule-based, and pattern-based programming paradigms. It is particularly popular in the physics community.

R is a freely available PSE targeted at statistical computing and data analysis. Many contributed R packages are available on the Comprehensive R Archive Network (CRAN).

Sage ia an open-source, Python-based PSE that builds on many other open-source packages. It has a browser-based notebook.

Project Jupyter (formerly known as IPython) is an open source project that includes a network protocol for interactive computing in any programming language, a browser-based notebook interface, and tools for sharing and converting these notebooks into multiple output formats, including HTML and PDF. This makes the Jupyter Notebook a fully-fledged PSE for Julia, Python, R, and other languages.

5 Programming Miscellany

We now focus on a variety of different aspects of programming that have a particular relevance to applied mathematics.

5.1 Pseudocode

In the early days of computing it was common to include a complete program listing in an article, as can be seen in the 1950s issues of the journal *Mathematical Tables and Aids to Computation*. This practice is now uncommon, not least because of the ease of distributing code over the web. It is now usual to describe in print the underlying algorithm in terms of a *pseudocode* that the author bases informally on the control structures and other syntax of a particular programming language (MATLAB being a common example). A good pseudocode combines precision, brevity, and readability. For examples of pseudocode see the article on Algorithms.

5.2 Abstraction

The mathematical concept of abstraction has proved to be important in programming, where it refers to separating concepts from implementation details. Subprograms take input arguments, carry out a computation, then return output. How they do it need not be known to the programmer who invokes them, so a subprogram is an abstraction of the computation it carries out. Abstraction applies to both procedures and data, and is used to the full in object-oriented programming.

5.3 Influence on Mathematics

While mathematics has had a strong influence on programming language design, programming languages have also influenced mathematics. We already noted that APL introduced the ceiling and floor notation. The array subscripting (or slicing) notation `A(i:j,p:q)`—used in Algol 68, MATLAB, and other languages to denote the subarray comprising the intersection of rows `i` to `j` and columns `p` to `q` of the two-dimensional array `A`—is now widely used in numerical linear algebra, especially in pseudocode.

In a 1928 paper, Kurt Hensel suggested the notation $A\backslash B$ for $A^{-1}B$ and A/B for AB^{-1}, but it did not catch on. Cleve Moler independently introduced the notation in MATLAB, and the term "backslash" is now commonly used to mean solving a matrix equation.

5.4 Notation for Expressions

In mathematics we normally write expressions in the conventional *infix notation* illustrated by `a + b(c-d)`, using parentheses and the usual precedence rules to specify the order of operations. In Lisp and related languages, the expression above is written

```
+ a * (b (- c d))        (1)
```

in which each arithmetic operator is followed by its two arguments. This *prefix notation* (also called Polish notation) is easier for computers to parse. The evaluation proceeds left to right, with the arguments of each operator evaluated recursively (in practice, using a stack), and no knowledge of the precedence of the operators is necessary.

The parentheses in (1) are not strictly necessary, but they are required in Lisp because operators can take multiple arguments: `+ 1 2 3` evaluates to `6`.

In *reverse Polish notation* (RPN) the operator follows rather than precedes the operands (as in the expression $n!$ for a factorial). The expression (1) is written

```
a b c d - * +
```

which is again evaluated left to right, with the variables a and b set aside until it is time to use them. An alternative way to write the expression that mingles the data and the operands is

```
c d - b * a +
```

RPN is used in the languages *Forth* and *PostScript*, and on HP pocket calculators.

5.5 Syntactic Peculiarities and Pitfalls

While there is much commonality between different languages, certain differences can catch the unwary programmer out. In most languages a single equals sign denotes an assignment: `x = 1`. A test for equality is written with a double equals in C and MATLAB: `if (x == y)`. If the test is written `if (x = y)` then in C this results in `y` being assigned to `x` and the `if` test being passed if `x` is nonzero, because `(x = y)` evaluates as a true Boolean expression. Algol and Pascal use `:=` for assignment, but this is not common in modern languages. R has two assignment operators, `<-` and `=`, of which only the former can be used anywhere in a program. The test for "not equal" is even more varied: `~=` in MATLAB, `!=` in C, R, and Python, `.NE.` in Fortran 77, `/=` in Fortran 90, and `<>` in Basic and Pascal.

A common operation is to increment a variable, which is typically done using a statement such as `x = x + 1`. Some languages provide a shorthand notation for this operation: in Python it is `x += 1` and in C, C++, and Java, `x++`. A subtlety is illustrated by the C code

```
i = 1; j = 1; a = i++; b = ++j;
```

which results in `a = 1` and `i = j = b = 2`, because the assignment is done before the incrementation with `i++` and after for `++j`.

Another aspect of syntax that varies among languages is operator precedence in expressions. An expression `a*b + c` is interpreted as $ab + c$ in most languages, but as $a(b + c)$ in APL, which does not have any operator precedence and always evaluates right to left. However, it is for relational and logical operators that differences are most common. An expression of the form `x or y and z` (with symbols such as `|` and `&` replacing `or` and `and` in many languages) can mean `x or (y and z)` or `(x or y) and z` depending on the language. In Lisp, expressions must be fully parenthesized, so they always have an unambiguous mathematical meaning.

5.6 Booleans

A Boolean, or logical, data type contains two possible values: true and false. Many languages denote these values `true` and `false`. Exceptions include Lisp (`t` and `nil`) and Fortran (`.true.` and `.false.`).

C does not have a Boolean data type and instead regards any nonzero numerical value as representing true and zero as representing false. In MATLAB, logical values are converted to 0 (for false) or 1 (for true) in numerical expressions, and this can be useful in a one-line expression such as

```
(exp(x) - 1 + (x == 0)) /( x + (x == 0))
```

which evaluates to $(e^x - 1)/x$ when $x \neq 0$ and to $1 = \lim_{x\to 0}(e^x - 1)/x$ when $x = 0$, avoiding what would otherwise be a division by zero.

5.7 Array Storage and Array Indexing

Fortran stores arrays in column major order, meaning that a two-dimensional array is stored sequentially in memory with the elements of the first column being followed by those of the second, and so on. C and many other languages store arrays in row major order. This difference is inconvenient when calling Fortran codes from other languages. Knowledge of the storage format is crucial, because for efficiency it is important to access elements of arrays in the order in which they are stored.

Programming languages differ as to the starting index for arrays. In Fortran and MATLAB, for example, arrays start at index 1 (`a(1), a(2), ...`), whereas in C and Python the first index is 0 (`a[0], a[1], ...`). Note that the type of brackets used for array indices, round or square, also varies, as illustrated. Mathematical descriptions of an algorithm may use 0 or 1 as the starting index, depending on the notation in effect.

The syntax for array slices also differs between languages. While in Fortran and MATLAB `a(i:j)` extracts `a(i), a(i+1), ..., a(j)`, in Python `a[i:j]` extracts `a[i], a[i+1], ..., a[j-1]`, so `a[j]` is omitted. These differences can be a cause of confusion and bugs. One needs to be aware of them and program with care.

5.8 Complex Arithmetic

Computations with complex numbers are ubiquitous in applied mathematics. From its earliest versions Fortran has had a complex data type that can be used in

expressions such as `a + b*c`, just like the real and double precision data types. In some other languages functions implementing complex arithmetic can be written, but then expressions must be converted to a sequence of function calls, such as `cadd(a,cmult(b,c))`. The PSEs mentioned above all support complex arithmetic, as do C (introduced in the 1999 standard), C++, Julia (which uses `im` rather than `i` for the imaginary unit), and Python (which uses `j` for the imaginary unit).

It cannot necessarily be assumed that the compiler or interpreter implements complex arithmetic in the most accurate and robust way. For example, if the modulus of a complex number is computed as $|a + \mathrm{i}b| = (a^2 + b^2)^{1/2}$ then the intermediate sum of squares can overflow even when $|a + \mathrm{i}b|$ is representable as a finite floating-point number. The possibility of overflow is easily avoided by evaluating $|a|(1 + (b/a)^2)^{1/2}$ when $|a| \geqslant |b|$ and an analogous expression when $|b| > |a|$. Operations such as complex division and evaluation of complex elementary functions are more difficult to implement reliably.

5.9 Variable Names

In mathematics, variable names are usually one letter: Greek or Roman, in lower case or upper case. Since Fortran introduced the possibility of variable names having more than one letter (albeit limited to six letters in Fortran 77 and earlier versions), multiletter names have been common. Due to the use of long variable names comprising several words joined together, several naming conventions have been introduced, illustrated by `endOfFile` or `EndOfFile` (camel case), `end-of-file`, and `end_of_file` (pothole case). Of course, which characters are allowed in variable names depends on the language. The use of long variable names is facilitated by text editors that allow autocompletion.

5.10 Floating-Point Semantics

Many mathematical relations fail to hold in floating-point arithmetic because of the effects of rounding errors. For example, `(a + b) + c` and `a + (b + c)` will in general evaluate to results differing at the roundoff level. Unfortunately, much more subtle issues can cause mathematical relations to break down. Intel x86 chips have 80-bit registers whose precision exceeds that of 64-bit double precision variables. After the assignment `x = 1.0/3.0` to a double precision variable `x`, a test `if x == 1.0/3.0` can return false with some optimizing compilers if `1.0/3.0` is temporarily stored in an extended precision register.

Some processors offer a *fused multiply-add* (FMA) instruction that evaluates an expression `x*y + z` with just a single rounding error, that is, the result is the exact value of `x*y + z` rounded to the target precision. The behavior of a program can then depend on the compiler in subtle ways. For example, the discriminant $b^2 - 4ac$ of a quadratic equation can evaluate as negative when $b^2 \geqslant 4ac$ if an FMA is used. These kinds of behavior make it very difficult to prove rigorous correctness results for computer programs executed in floating-point arithmetic.

5.11 Floating-Point Parameters

Programs that perform floating-point computation often need to use parameters of the floating-point arithmetic, such as the unit roundoff (typically in a convergence test) or the overflow level. Some languages, such as Fortran, provide direct access to these parameters via intrinsic functions. For those that do not, there are ways to compute them at run time, though these may not be entirely reliable when used with optimizing compilers.

5.12 High Precision Computations

The IEEE floating-point arithmetic standard defines single and double precision formats corresponding to about 8 and 16 significant decimal digits, respectively. Most programming languages support two floating-point data types that map onto these formats. A 2008 revision of the IEEE standard added a 128-bit quadruple precision format, which corresponds to about 32 significant decimal digits. Quadruple precision is not yet available in hardware, so arithmetic of precision higher than double must currently be provided in software.

In Fortran 90 and later versions of Fortran the availability of different precisions can be queried, through the `selected_real_kind` function. This allows access to quadruple precision if it is supported by the compiler.

A number of open source libraries are available that implement arbitrary precision floating-point arithmetic. The GNU MPFR library is a C library that provides correctly rounded arithmetic and mathematical functions, and it is used by Julia's BigFloat data type. The GNU MPC library builds on MPFR to handle complex arithmetic. Mpmath is a Python library for arbitrary precision floating-point arithmetic.

High precision arithmetic has many uses, including in experimental mathematics and for obtaining accurate solutions to ill conditioned problems. For a researcher developing or testing a numerical algorithm high precision provides a way to compute reference solutions that allow the accuracy of the algorithm to be tested.

5.13 Types

A number of subtle issues in programming languages revolve around the data type of a variable or expression: integer, floating-point, logical, string, and so on. Some languages, such as C and Java, require the type of a variable to be explicitly declared before an assignment is made to that variable. For example, in C the statement `double x = 1.1` both declares `x` to be a double precision variable and gives it the value 1.1. Some languages make specifying the type of a variable optional or not possible at all. Fortran uses implicit typing: if the type is not specified then a default type is assigned based on the first letter of the variable (integer for `i` to `p` and real otherwise). However, it is regarded as good practice to turn off this implicit typing with the statement `implicit none`. PSEs tend to determine the type at the point of assignment.

The type of a variable or expression might be fixed or it might be able to change during the execution of a program. For example, some languages allow a string to be added to a number and define the result to be either a string or a number. The terms *weakly typed* and *strongly typed* are often used in this context to characterize a language's type system, but these terms have no commonly agreed definition.

Type systems have an important influence on programs in at least two main ways. First, many programming errors are caused by variables (or constants) having an incorrect type. An apocryphal story tells of the loss of a 1960s NASA rocket due to the Fortran 66 software controlling the rocket having a line of the form `DO 10 k = 1.3` instead of the intended `DO 10 k = 1,3`, which starts a loop. The mistyping of a period for a comma in the former statement causes the Fortran compiler to interpret it as the assignment of 1.3 to the variable `DO10k`, since spaces are unimportant in Fortran 66 source code, and the implicit typing of Fortran causes the variable `DO10k` to be created with real type.

The second influence of a type system is on efficiency, since the speed at which a code executes will depend on how much the compiler or interpreter knows about the types of the variables. The computation of `x*y` will run much slower than it might if at run-time the types of `x` and `y` must be checked to decide whether to issue an integer multiplication or a floating-point multiplication instruction. Figure 3 illustrates the point, but in this instance the decision is made at compile time, with no loss of efficiency.

5.14 Complexity Analysis

Several measures of the complexity of a code have been proposed. They can be used to estimate the probability of bugs, the difficulty of testing, and the cost of maintenance of the code. The metrics apply to individual components such as functions, subroutines, and procedures, and a large complexity measure can be reduced by breaking the component into smaller pieces.

The simplest metric is the number of executable lines of source code. The *cyclomatic complexity*, or *McCabe complexity*, of a code is defined in terms of the directed graph that has nodes given by blocks of code containing no decisions or branches and edges corresponding to branches between nodes. The cyclomatic complexity is given by the formula edges − nodes + 2, and turns out to be equal to one plus the number of predicates (logical tests). The *Npath metric* is the number of possible execution paths through the code, which can be much larger than the cyclomatic complexity.

Tools are needed to compute these metrics. In MATLAB the function `checkcode` computes the cyclomatic complexity.

5.15 Formatting of Source Code

Mathematicians are used to having complete freedom in how they lay out their written mathematics on the page. Programming languages vary in their prescriptiveness of the layout of the source code. Most impose few restrictions and allow one to collapse a program block onto a single, very long line provided comments are removed and (if necessary) statement separators are added. When computers had small memories such a transformation would sometimes be done in order to save having to store the carriage return and line feed characters. Sometimes further code obfuscation is done in order to conceal the purpose of a code, for security reasons.

Fortran 77 requires code to lie between columns 7 and 72, with columns 1–5 reserved for statement numbers and column 6 for indicating a continuation line. These restrictions stem from the punched cards used to

enter programs into early computers and were removed in Fortran 90.

Some text editors provide automatic indentation tailored to the language being edited, and various pretty printing tools are available to format code for readability or to impose a particular house style. The use of such tools can aid debugging and make it simpler to compare different versions of a program with a diff command. Python is unusual in that it uses indentation to define if statements, for loops, while loops, and so on, whereas most languages use braces, brackets, or keywords to delimit code blocks.

5.16 Readability

Often there are several ways to write a piece of code. A balance needs to be struck between length of code, efficiency, and understandability. In C++, for an integer variable `n` one can compute the expression `2*n+1` as `n << 1 | 1`, where << is the bit-shift left operator and | is the bitwise or. The latter version is, however, rather inscrutable and may not be any faster than the former under a good compiler.

Sometimes one needs to make a variable cycle between several values. If the values are 0 and 1 then the assignment `n = 1 - n` flips between them and the purpose of the assignment is reasonably clear. Suppose, though, that we wish to make n take on the values 1, 2, 3, repeatedly. If we can find a polynomial p such that $p(1) = 2$, $p(2) = 3$, and $p(3) = 1$ then the assignment `n = p(n)` will do the trick. Such a p is a polynomial interpolant to the given data and the p of lowest degree is the quadratic $p(x) = -\frac{3}{2}x^2 + \frac{11}{2}x - 2$. However, the purpose of the assignment with p is not obvious and its correctness is not trivial to check. An `if` statement of the form

```
if n == 1
   n = 2
elseif n == 2
   n = 3
else
   n = 1
end
```

does the job in a more transparent fashion. Alternatively, an assignment replacing `n` by $(\mathtt{n} \bmod 3)+1$ could be used, supplemented by a comment explaining its purpose.

5.17 Structured Programming

In Fortran a `go to` statement causes a jump to a labeled statement anywhere in the program. In 1968 Edsger Dijkstra wrote a letter to the editor of the journal *Communications of the ACM* in which he claimed that the use of `go to` statements, which were very common in Fortran 66 programs, represented poor programming practice. The letter was published with the title “Go to statement considered harmful”. The notion of *structured programming* subsequently became popular. Structured programming enforces a logical structure on the program that makes it easier to understand and modify, through the use of certain canonical control structures together with modular composition of programs. A long 1974 paper by Donald Knuth titled “Structured programming with `go to` statements” presents a balanced analysis of the pros and cons of `go to` statements.

5.18 Literate Programming

In the 1980s Knuth championed the idea of *literate programming*, in which a document contains a combination of source code and documentation for the code (in TeX format) and both the code and the documentation can be generated from it. He used this approach to great effect in writing TeX and associated programs using his WEB system (which has no connection with the worldwide web, which it predates). Nowadays, literate programming is mainly used in two forms. In the first, documentation is embedded in comment lines of a program’s source code and documentation generation tools are used to extract it to HTML, PDF, etc. In the second form, a document contains code that carries out the computational experiments needed for a paper, and a separate “preprocessor” executes the code and inserts its output (numeric or graphical) back into the source document. This approach facilitates reproducible research and is typically done with “weave” tools available for R and Python or in Emacs Org mode.

5.19 Interoperability

Interoperability refers to the ability to call a program written in one language from a program written in a different language. Historically, the degree of interoperability that is available has depended on which operating system and compiler is in use, as well as on the languages themselves. Even when cross-language calls

are possible there are pitfalls to watch out for, such as the potentially different ways in which multidimensional arrays are stored in different languages (see section 5.7). There is a strong trend to mixed language programming, encouraged by languages such as C++, Julia, and Python that have been designed with interoperability in mind, by the support provided in PSEs for calling or being called by another language, and by languages that are built on the same virtual machine (such as Java, Scala, and Clojure).

5.20 Domain-Specific Languages

A domain-specific language (DSL) is a language focused on a particular problem domain, examples being HTML for web pages, SQL for databases, and TEX for mathematical typesetting. An important benefit of a DSL is that it can allow programming at a high level of abstraction that fully exploits knowledge of the problem domain and thereby reduces the total time to deliver a solution to a problem.

Applied mathematics has a variety of DSLs, and these often involve symbolic manipulation as part of the code generation process. The *General Algebraic Modeling System* (GAMS) is a high-level modeling system for mathematical optimization. It includes a DSL in which optimization problems of several different types can be specified. A number of DSLs are associated with software for solving partial differential equations. For example, the *Unified Form Language* in the FEniCS project is a DSL for finite element discretizations, implemented as a Python module.

DSLs for plotting graphics are plentiful, even being built on top of other DSLs (for example, the various graphics packages for LATEX).

5.21 Translation Between Languages

In mathematics we are used to translating between different notations, and moving from one space or basis to another. It is natural to ask whether a program can be transformed from one language to another without any change in its behavior. One reason for wanting to do so is to convert programs that were written many years ago but are still used today (legacy codes) into a more modern language. Such translation tools are available, but they are used out of necessity rather than as a standard tool. A tool called f2c written at Bell Labs in the 1990s could convert Fortran 77 codes to C, though the resulting code was not meant to be readable by humans. There are more recent tools for converting Fortran to C++ that produce more readable code.

Table 1 Extract from the TIOBE Programming Community Index for February 2015. Clojure, Forth, Mathematica, and OpenCL are all ranked in the range 51–100.

Language	Rank	Language	Rank
C	1	Pascal	19
Java	2	PostScript	24
C++	3	Fortran	31
Python	8	Lisp	32
Visual Basic	9	Scheme	38
MATLAB	17	Scala	41
R	18	PL/I	45

5.22 Popularity of Languages

An interesting question is which are the most popular programming languages. This question is both hard to define precisely and hard to answer. One attempt is provided by the TIOBE Programming Community Index ("www.tiobe.com"), which is produced once a month based on “the number of skilled engineers world-wide, courses and third party vendors”, as found via popular search engines. Table 1 shows a ranking of most of the languages mentioned in this article. These rankings are quite volatile and should not be taken too seriously, but an interesting implication is that old languages such as Fortran and Lisp continue to compete with their younger counterparts.

5.23 Language of the Future

An old joke goes “I don’t know what language we’ll be using in 50 years time, but it will be called Fortran.” Fortran has been under attack since the 1960s but shows no signs of dying, as noted in the previous subsection. The frequent revisions to the Fortran standard have kept the language up to date, while the huge amount of legacy code means that in many applications it is difficult or impossible to switch to alternative languages. The improved interoperability of languages and compilers enables binaries of compiled Fortran libraries such as LAPACK and the commercial NAG Library to be readily called from other languages and even Excel spreadsheets. Perhaps the future is inherently multilingual, with programs being written in a modern language such as C++ or Python and calling kernels written in C, Fortran, or assembly language tuned for particular processors by the manufacturer.

One thing we can be sure of is that new languages will continue to be developed, each trying to combine the best features of existing languages with new ideas

that resonate with developments in hardware and software. However, it is important that language designers remember the lessons of the past and contemplate the comment of Tony Hoare about Algol 60: “Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.”

6 Further Reading

The following list is very selective and merely provides a starting point for further exploration.

Abelson and Sussman (1996) is a classic introduction to programming based on Scheme that emphasizes ideas such as abstraction and recursion over syntax. It has many interesting mathematical examples, including symbolic differentiation.

The longevity of Fortran, with its multiple revisions, is such that its history, as told by Metcalf (2011), provides a prism into the history of programming languages.

The Turing Award of the Association for Computing Machinery (ACM) is an annual award that is to computer science what the Fields Medal is to mathematics. The book of lectures from the first twenty years of awards is full of insights into programming languages. It includes lectures by, among those mentioned in this article, Backus, Dijkstra, Iverson, Knuth, McCarthy, Ritchie, and Wirth.

An excellent source for the history of programming languages (and computing) is the journal *IEEE Annals of the History of Computing.*

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* Second edition, The MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-51087-1.

[2] ACM. *ACM Turing Award Lectures: The First Twenty Years, 1966–1985.* Addison-Wesley, Reading, MA, USA, 1987. xviii+483 pp. ISBN 0-201-54885-2.

[3] Jon L. Bentley. *More Programming Pearls: Confessions of a Coder.* Addison-Wesley, Reading, MA, USA, 1988. viii+207 pp. ISBN 0-201-11889-0.

[4] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style.* Second edition, McGraw-Hill, New York, 1978. xii+168 pp. ISBN 0-07-034207-5.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Second edition, Prentice-Hall, Englewood Cliffs, NJ, USA, 1988. xii+272 pp. ISBN 0-13-110362-8.

[6] Donald E. Knuth. *Selected Paper on Computer Languages.* CSLI Lecture Notes Number 139. Center for the Study of Language and Information, Stanford University, Stanford, CA, USA, 2003. xvi+594 pp. ISBN 1-57586-382-0.

[7] Michael Metcalf. The seven ages of Fortran. *Journal of Computer Science and Technology*, 11(1):1–8, 2011.

[8] Bjarne Stroustrup. *The C++ Programming Language.* Fourth edition, Addison-Wesley, Upper Saddle River, NJ, USA, 2013. xiv+1346 pp. ISBN 978-0-321-56384-2.