

## A brief history of notations for algorithms

January 19 and 21, 2009

The study of algorithms began with and is motivated by the human desire for answers to questions. For practical reasons, people prefer answers that are, in the first place, true (since one's actions are more likely to have satisfactory results if one understands the world as it is) and, secondly, arrived at by objective methods, which make it possible to secure the agreement and cooperation of others (since they can apply the same objective methods to confirm the truth of one's answers).

For instance, at some point before history began, people discovered an accurate and objective way to determine which, if either, of two groups of discrete things is more numerous than the other. One arranges the members of the groups in pairs, each pair comprising a different member from each group, until the members of one group are exhausted. If the other group is not yet exhausted, it is more numerous; otherwise, neither group was more numerous.

In many cases it is easy to perform the pairing operation correctly and easy to check that someone else is performing it correctly, so that the same answer is likely to be obtained no matter who performs the pairing and regardless of the circumstances in which it is conducted. Moreover, one needs neither great skill nor great faith in mathematics to understand why the method gives the right answer. So this ritual provides an impartial and commonly acceptable way to settle disputes.

In cases where the groups to be compared cannot be brought together to be paired off, one can pair off the members of either group with counters, such as pebbles, shells, or slashes on a wooden stick, and then carry the counters to the other group and pair again, with each counter representing the member of the absent group with which it was formerly paired. This method, too, produces reliable, credible, and reproducible answers.

Initially, such rituals arise only when the need to determine which of two groups of things is more numerous occurs frequently. They arise because sufficiently experienced people can perceive that the individual comparisons have a common structure—that the question whether Kohath has more sheep than Merari, and the question whether the tribe of Enan or the tribe of Ocran is larger, and the question whether Bezaleel or Aholiab possesses more pieces of silver plate, are all instances of the same problem, the problem of determining which of two groups of things is more numerous. The pairing ritual is a *general* way to address this problem, to solve *any* instance of it.

Whenever a number of questions have a similar structure, they are instances of the same problem, and frequently there is a common method by which any of them can be answered. Given a question of the form “Which, if either, of two groups of discrete objects is more numerous?” the pairing ritual tells you, step by step, what to do in order to answer that question. And that's what an algorithm is: an effective, step-by-step, computational method for solving any instance of a specified problem.

Perhaps there was a time when pairing was the only form of computation. However, if one performs the pairing ritual often, especially with the help of counters, the configurations of counters become familiar and gradually acquire identities and names of their own:

one begins to recognize them as *numbers*. It then becomes possible to ask and answer a slightly more abstract question: How numerous is this group? How many things are in it? Furthermore, once one is accustomed to asking and answering this question, it becomes unnecessary to carry the counters around in order to compare groups. Instead, one need only record or remember the number of things in one group and use the same ritual for counting the things in the other group. The abstract sequence of numbers takes the place of physical counters.

In spite of this higher level of abstraction, counting remains an objective procedure. It delivers true answers that can be checked and confirmed by skeptics, and everyone who understands the counting ritual can agree on those answers. Counting, too, is an algorithm.

Next, having learned to perceive numbers as individual entities, a person can observe that they also stand in relationships and exemplify repeated structures and patterns—that there are general truths about numbers themselves, and objective ways to find those truths. On this basis one can formulate other questions and answer them by performing more elaborate computational rituals—addition, for instance, or the extraction of cube roots. Unlike pairing, which anyone can do with a little attention and patience, the rituals of arithmetic require training and practice. Some of them are intricate, long, and hard to understand. In such cases it is possible to separate the ritual from the understanding that grounds it, so that accepting the answers it delivers becomes an act of faith rather than reason. Also, faults in performance are common, even for well-trained practitioners.

Successive steps of abstraction and generalization lead from positive integers to ratios, negative numbers, irrational numbers, complex numbers, quaternions and octonions, Conway numbers, vectors, matrices, sequences, sets, functions, graphs, groups, fields, categories, and so on. In these rich mathematical universes we also find models and encodings for all kinds of non-numeric and even non-mathematical data, from individual Booleans and characters to texts, graphics, sounds, animations, and the like. Since we can specify the encoding processes themselves as computational rituals, the notion of computation that we now use is no longer tied to numbers, but applies to the processing of data of any kind.

With each new development, new abstractions made it possible to ask new questions, and new methods of determining the answers to those questions were developed. Until about 1890, however, four related difficulties or limitations severely constrained the performance of computational rituals:

- In the absence of any unambiguous general-purpose notation for recording algorithms, it was often difficult to figure out exactly how to enact a particular computation. Descriptions of algorithms in ordinary prose often omitted details, failed to explain how to deal with exceptional cases, or required the performer to guess how to proceed at key points. (Consider, for instance, the process of computing the next digit of a quotient in the long-division ritual.)
- Since an algorithm could be and often was separated from the understanding that grounded and justified it, performers habitually followed rules on faith (“just because it works”). Unfortunately, like other kinds of faith, computational faith is extremely error-prone: If you don’t know exactly what you’re doing or why you’re doing it,

there's a much greater chance that you're doing it wrong and that the results will be unsatisfactory.

- To make matters worse, there was frequently no simple way to detect a fault in the performance of an intricate computational ritual or, having detected a fault, to remedy it.
- Performing an algorithm that involved a large number of steps or a large quantity of data required extraordinary patience, care, and tenacity and even so usually yielded incorrect results. In the middle of the nineteenth century, a monomaniac hobbyist named William Shanks devoted twenty years of his free time to the computation of a high-precision value for  $\pi$ , obtaining 707 digits after the decimal point. This stood as a sort of record for a single computation until the 1950s, when it was discovered that he made a mistake that affected the 528th and all subsequent digits. The record for a collection of related computations performed without mechanical aids is probably the 1880 census of the United States, which took seven or eight years to complete and was almost certainly riddled with incorrect results.

In our time, the invention and development of stored-program computers have lifted the last of these constraints. We can now expect to compute 707 digits of  $\pi$  in a second or less, while the 1880 census computations, which would probably be I/O-bound, might take as much as a minute, disk drives being slower than processors. A “long” computation today would be something like computing forty billion digits of  $\pi$ , and we would expect all forty billion to be correct. A “large” data set today would be measured in terabytes.

We have made less progress with the second and third difficulties. In fact, our increasing reliance on machines to perform computational rituals has exacerbated them. Millions of people now have calculators with square-root keys; few of them could either describe or perform an algorithm for computing a square root without mechanical aids, and only a fraction of those could explain why their algorithm works. Calculators are so reliable that we seldom notice their limitations, and when they contain defects that produce incorrect answers we are apt not to notice the failures even when they would be obvious to a someone performing the same computation manually. (And when we are convinced that a calculator has failed, there is nothing to do about it except buy a new calculator.)

We have come to rely in daily life on the correctness of immense numbers of intricate computations that we do not understand. On one hand, this reliance reflects and underscores the fragile complexity of our civilization, with its extreme division of labor. It is enough that just a few people really understand the computation of square roots and can embed their understanding in calculating machines. Everyone else in the world who can afford a calculator then gets all the benefits of being able to compute square roots while remaining free to understand and do other useful things instead. On the other hand, those who take square roots completely on faith enter a kind of intellectual slavery, a tyranny imposed by the calculators and their makers—benevolent, or at least mutually profitable, when the computational rituals are correctly enacted, but as despicable as any other kind of tyranny when they are not.

The first of the four constraints that I mentioned above, however, has now been almost eliminated by the invention, development, and widespread use of high-level programming

languages. It is now possible, indeed common, for the creator of an algorithm to express it exactly, completely, and unambiguously in the form of a program. There are still some difficulties. For instance, some authors naively and erroneously believe that floating-point representations obey the same arithmetic laws as the real numbers they approximate, and some fail to distinguish between integers, on one hand, and integers modulo some power of two on the other. But the standards of expression are so much higher today than they were in the pre-computer era that I think we can boast of having almost solved this difficult intellectual problem.

To see how far we've come, consider Euclid's algorithm for finding the greatest common divisor of two natural numbers. Here, first, is how Euclid expressed it:

*Given two numbers not prime to one another, to find their greatest common measure.*

Let  $AB$ ,  $CD$  be the two given numbers not prime to one another.

Thus it is required to find the greatest common measure of  $AB$ ,  $CD$ .

If now  $CD$  measures  $AB$ —and it also measures itself— $CD$  is a common measure of  $CD$ ,  $AB$ .

And it is manifest that it is also the greatest; for no greater number than  $CD$  will measure  $CD$ .

But, if  $CD$  does not measure  $AB$ , then, the less of the numbers  $AB$ ,  $CD$  being continually subtracted from the greater, some number will be left which will measure the one before it.

For an unit will not be left; otherwise  $AB$ ,  $CD$  will be prime to one another, which is contrary to the hypothesis.

Therefore some number will be left which will measure the one before it.

Now let  $CD$ , measuring  $BE$ , leave  $EA$  less than itself, let  $EA$ , measuring  $DF$ , leave  $FC$  less than itself, and let  $CF$  measure  $AE$ .

Since then,  $CF$  measures  $AE$ , and  $AE$  measures  $DF$ , therefore  $CF$  will also measure  $DF$ .

But it also measures itself; therefore it will also measure the whole  $CD$ .

But  $CD$  measures  $BE$ ; therefore  $CF$  also measures  $BE$ .

But it also measures  $EA$ ; therefore it will also measure the whole  $BA$ .

But it also measures  $CD$ ; therefore  $CF$  measures  $AB$ ,  $CD$ .

Therefore  $CF$  is a common measure of  $AB$ ,  $CD$ .

I say next that it is also the greatest.

For, if  $CF$  is not the greatest common measure of  $AB$ ,  $CD$ , some number which is greater than  $CF$  will measure the numbers  $AB$ ,  $CD$ .

Let such a number measure them, and let it be  $G$ .

Now, since  $G$  measures  $CD$ , while  $CD$  measures  $BE$ ,  $G$  also measures  $BE$ .

But it also measures the whole  $BA$ ; therefore it will also measure the remainder  $AE$ .

But  $AE$  measures  $DF$ ; therefore  $G$  will also measure  $DF$ .

But it also measures the whole  $DC$ ; therefore it will also measure the remainder  $CF$ , that is, the greater will measure the less: which is impossible.

Therefore no number which is greater than  $CF$  will measure the numbers  $AB$ ,  $CD$ ; therefore  $CF$  is the greatest common measure of  $AB$ ,  $CD$ . Q. E. D.

Now, this version does have one merit: The proof of the correctness of the algorithm is provided along with the description of how it works. By modern standards, it's not a very good proof; Euclid's typical way of verifying a recursive algorithm was to work through the cases of  $n = 0$  and  $n = 2$ , relying on the reader to recognize the pattern and to infer that it could be extended to any natural number. But Euclid does offer a reason to believe that the algorithm is correct, and in fact it is pretty straightforward to tighten up his reasoning to get a proper proof by mathematical induction.

On the other hand, there are a lot of serious problems with this presentation. Some are just due to differences between Euclid's way of conceiving numbers and ours; for him, numbers are line segments that are exact multiples of some standard unit, and dividing one number by another is measuring out copies of one line segment against another until you can't fit in any more copies. As a result, he uses geometrical language that seems awkward and wordy rather than the arithmetic or algebraic language that would be more natural to the problem.

Moreover, the unit itself is not a number in Euclid's terminology; numbers are  *multiples* of the unit—two or more. This is why Euclid restricts his proposition to “numbers not prime to one another,” that is, numbers that have a common divisor greater than 1. Of course, the same method works perfectly well when the numbers *are* prime to one another, giving what we would regard as the correct answer, namely 1. Euclid would not have regarded this as an answer at all, since 1 is not a number.

Perhaps the most serious weakness of Euclid's presentation is that it relies on references to unpublished diagrams. Euclid expected that human teachers would present his proofs to students, who would in turn present them to other students, and so on down through the generations. Everyone in the chain would learn the diagrams from his teacher and teach them to his students. The invention of the printed book disrupted this chain, because it led people to expect to learn things without the presence of human teachers. This forced Euclid's editors to integrate diagrams into the text, which is almost unintelligible without them.

Also, although Euclid realizes the utility of having symbolic names like  $AB$  and  $G$  for the numbers that he is working with, he has no symbols for relationships between numbers or operations on numbers, and this makes his exposition unbearably wordy. Compare this to the way we'd write his algorithm in a modern programming language:

```
(define greatest-common-divisor
  (lambda (ab cd)
    (if (zero? cd)
        ab
        (greatest-common-divisor cd (remainder ab cd)))))
```

The use of symbolic names for procedures as well as for numbers enables us to pare off and discard almost everything in Euclid’s presentation, retaining only the two or three key ideas. And this is for an easy algorithm. Think how difficult it must have been to predict an eclipse or to calculate the date of Easter using such methods.

Let’s now trace the history of the development of notations for expressing algorithms, identifying some of the most important achievements.

Charles Babbage designed the first general-purpose, programmable calculating machine in 1837. It was called the Analytical Engine. Babbage’s funding ran out before he completed a working implementation. He and his associate Ada Augusta Byron, Countess Lovelace, developed a semiformal notation for machine states and execution traces, which can be regarded as incorporating a notation for programming.

Alan Turing’s paper “On computable numbers with an application to the Entscheidungsproblem” (*Proceedings of the London Mathematical Society* 2 [1936]: 42, 230–265) includes a general and completely formal notation for defining a special-purpose automaton for any algorithm. Such an automaton definition can also be regarded as a program to be executed interpretively by a general automaton simulator; since simulation is an algorithmic process, this simulator can be another automaton of the same sort, defined in the same way. Turing’s automata were mathematical entities, not realized in hardware.

One early electronic computer, the ENIAC, developed by researchers at the Moore School for the U.S. Army in 1943–1945, was programmed by plugboard. The arithmetic instructions were entered by setting several thousand two-position switches, but the flow of control from instruction to instruction was determined by a different method: The machine had large panels containing, in all, a few thousand sockets, in an arrangement similar to an old-fashioned telephone switchboard, and cables were inserted into sockets to guide the course of the computation. A “program” was a cabling diagram.

The first execution of a program on a *stored-program* electronic computer took place at Manchester University on June 21, 1948; it factored integers. The computer was called the Manchester Mark I (not to be confused with the earlier Mark I at Harvard University, which was not electronic and did not store programs internally). The Manchester Mark I used cathode-ray tubes, like the picture tubes of old-fashioned television sets, as *storage devices* (not just for output). The program seems to have been loaded by setting switches, and it is possible that no systematic notation was used for coding.

The term “coding” for the preparation of computer programs appeared about 1950 or 1951, shortly after mechanisms for loading programs from paper tape or punched cards were invented. The basis for the term is that writing machine instructions from a detailed algorithm is like enciphering a message in a complicated cryptological system, and the resulting instructions are essentially unintelligible. Initially, a human programmer worked out the exact sequence of instructions that he wanted the computer to execute; the “source code” for the program was this sequence of machine instructions, as punched into the tape or cards.

Characteristically, early machine programmers wanted to automate their own jobs first. The earliest non-numerical programs were assemblers. An assembler reads in a representation of a program in which each operation that the processor can perform has

an alphanumeric name, such as `ADD` or `BRZ`. The programmer can also select alphanumeric names for particular machine addresses and data values. The assembler goes through such a program representation and substitutes machine-language equivalents for each of the names it encounters. In the simplest assemblers there is a one-to-one correspondence between assembly-language instructions and machine instructions. However, assemblers rapidly accumulated features such as automatic computation of the machine addresses at which instructions will be stored, pseudo-instructions that direct the assembly process instead of being converted into machine instructions, and macro definitions.

The first high-level programming language was FORTRAN, the earliest versions of which were developed at IBM by a team headed by John Backus between 1953 and 1958. (The best and most complete of the documents accompanying the work is dated 1957, so this is often cited as “the” date of the appearance of FORTRAN.) FORTRAN was the first programming language to provide evaluation of arithmetic expressions of arbitrary complexity, array subscripting, iterative and conditional control structures, independently compiled subroutines, and formatted I/O.) The design of FORTRAN strongly influenced many subsequent programming languages, specifically BASIC and PL/I.

Algol 60 (developed between 1957 and 1962) was the first language to receive a completely formal syntax definition. It was the first free-format language, taking a block (declarations plus executable statement) as the natural unit of composition and allowing arbitrarily complex nesting of blocks. It was the first language to have explicit typing of variables; the first to provide a mechanism by which the programmer could control the scope of identifiers; the first to permit recursive procedures and functions; and the first to permit arrays whose sizes are determined at run time. Algol 60 also strongly influenced many later languages, most notably Pascal, Algol 68 (which however differs from Algol 60 much more radically than the name might suggest), C, Scheme, Modula-2, Ada, and Java.

COBOL (developed between 1959 and 1961) was designed specifically as a language for commercial data processing applications. It introduced two important ideas in language design: records (sometimes called structs by C programmers or tuples by theorists) as user-defined data types, and standardization across machines and implementations. COBOL, as it developed, also consistently provided better facilities for the description and manipulation of files than other languages, right up to the mid-1990s, when the implementers of Java libraries finally figured out how to do an even better job.

PL/I (developed between 1964 and 1969) was the first language to receive a completely formal semantic definition, written in a notation called the Vienna Definition Language. It is a “kitchen-sink” language, alleged by its designers to combine the best features of FORTRAN (which contributed mainly the syntax of expressions and statements), Algol 60 (the notions of block structure and recursion), and COBOL (the facilities for definition of data types). In retrospect, this eclectic approach was not a good idea. PL/I had such a large and diverse core that every programmer effectively selected her own subset of it, which impeded communication between programmers and made programs harder to debug. The language also suffers from the absence of a clear conceptual model; its virtual machine is neither a classical von Neumann machine like FORTRAN’s, nor a run-time stack machine like Algol 60’s.

All of the preceding languages are based on the imperative model of computation. All

of them take statements or groups of statements as the fundamental building blocks of programs; expressions are evaluated only incidentally, as parts of statements. LISP (designed and first implemented in 1959 and 1960) is an expression-based language embodying the functional model of computation. Every language construct is thought of as producing a value, and the closest analogues of statements are expressions whose values are discarded, which are regarded as function calls that have desirable side effects.

John McCarthy, the original designer of LISP, gave a nearly complete formal specification of the semantics of LISP, which however cannot quite be called a definition because the formalism used in the specification is LISP itself. (There is another way of describing exactly what McCarthy did: He wrote a LISP interpreter in LISP.) Originally LISP had only two data types, and users implemented all others in terms of these in application programs; however, modern dialects of LISP have much more complicated type systems. LISP is also one of the earliest languages not to have been designed specifically for numerical applications; its intended problem domain was the field now known as artificial intelligence.

Comit (designed between 1957 and 1961) and SNOBOL (developed between 1962 and 1967, at which time it was more or less frozen as SNOBOL4) were the first languages to provide a general character-string data type and appropriate operations. Both languages have serious design flaws: SNOBOL's only control structures are conditional and unconditional `gotos`, and Comit lacks the notions of string-valued variables and assignments. However, in many other ways, SNOBOL4 is an ingenious and strikingly powerful language. In addition to strings, it provides patterns, dynamically sized arrays, associative tables, and dynamically constructed records as data types. As in most implementations of Scheme, types are not declared or checked during translation, but only during execution.

Implementation of SNOBOL4 on a variety of machines was simplified by the designers' strategy of writing the first translator in the SNOBOL Implementation Language, a language consisting entirely of assembler macro instructions. To get SNOBOL working on a new machine, an implementer would write appropriate macro definitions for the SIL instruction set, slap them on the front of the existing SNOBOL4 implementation, and assemble the result. SNOBOL4 has influenced some later languages, notably Icon.

APL (designed between 1961 and 1967) is, loosely speaking, a statement-based language with extremely primitive control structures, compensated for by an amazing variety of operators, almost all of which can operate on vectors, matrices, and arrays of higher dimensions as well as on numbers or characters (so that, for example, the plus sign stands for vector and matrix addition as well as scalar addition). The theory is that you don't need loop control structures, since looping over the elements of an array (of any dimension!) is automatic. An APL implementation typically provided a built-in, interactive programming environment, as did BASIC, which was designed slightly later (between 1964 and 1970).

All of the languages cited above use a primary model of program development in which any application program consists of a single compilation unit, except possibly for the invocation of precompiled library functions. The idea that a single original application might be created in several separate pieces that are subsequently linked together emerged gradually during the 1960s, in connection with the shift of technology from punched cards

to files stored electronically on tapes or disks. By the time C appeared (developed at Bell Labs, between 1968 and 1970), it was starting to be conventional to divide a large program into many compilation units and to make a clean distinction between a function's interface and its implementation.

APL is one of a number of languages that derive much of their distinctiveness from an obsessive focus on one type of data structure (in APL's case, the array). Early versions of SNOBOL focussed similarly on strings (in SNOBOL1, addition was a string operation!), early versions of LISP on lists. The FORTH language sees all of its operations as applying to a stack (more precisely, to a pair of stacks); PostScript is another language that uses the same model (although it specializes in page description, PostScript can actually be used as a more general programming language).

Simula (designed between 1962 and 1967 at the Norwegian Computing Center by Ole-Johan Dahl and Kristen Nygaard) was the earliest of the “object-oriented” languages, which share a similar obsession with the theme of data abstraction and encapsulation. This idea was perhaps most fully realized in Smalltalk-80 (designed between 1972 and 1980 at Xerox PARC). In Smalltalk-80, the object model of computation extends all the way down to the primitive values; when the expression  $5 + 7$  is evaluated, the Smalltalk-80 processor sees the message “add 7 to yourself” being sent to the object 5, which interprets the message, performs the operation, and returns the result 12 to the sender.

Many object-oriented languages, however, have mixed models of computation, typically grafting constructions supporting the object-oriented model onto an underlying imperative/procedural base, as in C++ and Visual BASIC, and to a lesser extent in Java, all of which were designed and implemented after Smalltalk-80, with the explicit goal of making it easy for programmers accustomed to the imperative model to make the transition to object-oriented software development.

The Prolog programming language, designed in the early 1970s, exemplifies yet another model of computation. Prolog is a declarative language; a Prolog program consists (mainly) of assertions about logical relationships rather than instructions, and the operations provided by the language are various ways of connecting these assertions and guiding the run-time system as it makes inferences from them. The Prolog virtual machine presupposes that the programmer will usually be indifferent about the flow of control, the order in which the operations will be performed. Small Prolog programs often contain no explicit indications of control flow at all, and even in large programs it is mostly concealed both from the programmer and from the program reader, just as a language like Java conceals the machine addresses of variables.

Scheme first appeared in 1975 as a kind of experimental dialect of LISP that embodied several original design decisions, most notably the uniform treatment of procedures as values and guaranteed tail-call optimization (which made it possible, for the first time, to completely eliminate explicit iteration from the language, replacing it with seemingly recursive procedure calls). Scheme also demonstrated that a small core language with a *very* small number of general, orthogonal expression types can provide the basis for a useful programming language. Fairly early on, Scheme incorporated into its standard a formal definition of its semantics, using a notation which, though difficult, was much easier to learn and use than the Vienna Definition Language.

ML is another functional-programming language developed about the same time as Scheme (by a team headed by Robin Milner at the University of Edinburgh). ML was the first programming language I know of that included automatic type inference; it allows users to specify the types of expressions, but the language processor is usually able to infer the types if they are omitted and to detect type errors before execution begins.

Modern languages that are radically different in design from those in the major families identified above tend to be special-purpose languages tailored to their applications (e.g., real-time equipment control, typesetting and page description, report generation). In particular, the rise of the Internet as an arena for computation has led to the rapid development of scripting languages and frameworks, designed for the quick creation and deployment of small applications and Internet services) (ECMAScript, Perl, Python, PHP, Ruby). Programs written in these languages tend to be error-prone, difficult to maintain, and slow in execution but their flexibility, their widespread availability and the large libraries of useful code written in them sustain their use.

Still another family of languages, or more often of extensions to existing languages, is designed to cope with radically different modern computer architectures—specifically, with multi-processor machines, on which operations can be performed in parallel. One approach is to lift the notion of a process from the operating-system level into the high-level language, as in Ada (designed between 1977 and 1979) or Java (designed between 1991 and 1995). Another idea is to allow multiprocessing at the statement level: Just as C provides a sequential compound statement, framed by left and right braces, some languages provide a parallel compound statement, the parts of which can be distributed to different processors for execution in any order.

In a one-semester course, we won't have time to examine the implementation of all the language features described here, but we'll look closely at a lot of the big and influential ideas: expression evaluation, identifiers and scope, state and assignment, control structures, type systems, module system, and how the various models of computation are implemented. As we do so, keep in mind the question of how these various language features simplify and clarify the expression of algorithms and supply the conceptual framework for the effective development of software.