1974 Turing Award Lecture

Computer Programming as an Art

DONALD E. KNUTH

[The Turing Award citation read by Bernard A. Galler, chairman of the 1974 Turing Award Committee, on the presentation of this lecture on November 11 at the ACM Annual Conference in San Diego.]

The A. M. Turing Award of the ACM is presented annually by the ACM to an individual selected for his contributions of a technical nature made to the computing community. In particular, these contributions should have had significant influence on a major segment of the computer field.

"The 1974 A. M. Turing Award is presented to Professor Donald E. Knuth of Stanford University for a number of major contributions to the analysis of algorithms and the design of programming languages, and in particular for his most significant contributions to the 'art of computer programming' through his series of well-known books. The collections of techniques, algorithms, and relevant theory in these books have served as a focal point for developing curricula and as an organizing influence on computer science."

Such a formal statement cannot put into proper perspective the role which Don Knuth has been playing in computer science, and in the computer industry as a whole. It has been my experience with respect to the first recipient of the Turing Award, Professor Alan J. Perlis, that at every meeting in which he participates he manages to provide the insight into the problems being discussed that becomes the focal point of discussion for

Author's present address: Fletcher Jones Professor of Computer Science, Stanford University, Stanford CA 94305.

the rest of the meeting. In a very similar way, the vocabulary, the examples, the algorithms and the insight that Don Knuth has provided in his excellent collection of books and papers have begun to find their way into a great many discussions in almost every area of the field. This does not happen easily. As every author knows, even a single volume requires a great deal of careful organization and hard work. All the more must we appreciate the clear view and the patience and energy which Knuth must have had to plan seven volumes and to set about implementing his plan so carefully and thoroughly.

It is significant that this award and the others that he has been receiving are being given to him after three volumes of his work have been published. We are clearly ready to signal to everyone our appreciation of Don Knuth for his dedication and his contributions to our discipline. I am very pleased to have chaired the Committee that has chosen Don Knuth to receive the 1974 A. M. Turing Award of the ACM.

When *Communications of the ACM* began publication in 1959, the members of ACM's Editorial Board made the following remark as they described the purposes of ACM's periodicals [2]: "If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected." Such a goal has been a continually recurring theme during the ensuing years; for example, we read in 1970 of the "first steps toward transforming the art of programming into a science" [26]. Meanwhile we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it "computer science."

Implicit in these remarks is the notion that there is something undesirable about an area of human activity that is classified as an "art"; it has to be a Science before it has any real stature. On the other hand, I have been working for more than 12 years on a series of books called "The *Art* of Computer Programming." People frequently ask me why I picked such a title; and in fact some people apparently don't believe that I really did so, since I've seen at least one bibliographic reference to some books called "The *Act* of Computer Programming."

In this talk I shall try to explain why I think "Art" is the appropriate word. I will discuss what it means for something to be an art, in contrast to being a science; I will try to examine whether arts are good things or bad things; and I will try to show that a proper viewpoint of the subject will help us all to improve the quality of what we are now doing.

One of the first times I was ever asked about the title of my books was in 1966, during the last previous ACM national meeting held in Southern California. This was before any of the books were published, and I recall having lunch with a friend at the convention hotel. He knew how conceited I was, already at that time, so he asked if I was

going to call my books "An Introduction to Don Knuth." I replied that, on the contrary, I was naming the books after *him*. His name: Art Evans. (The Art of Computer Programming, in person.)

From this story we can conclude that the word "art" has more than one meaning. In fact, one of the nicest things about the word is that it is used in many different senses, each of which is quite appropriate in connection with computer programming. While preparing this talk, I went to the library to find out what people have written about the word "art" through the years; and after spending several fascinating days in the stacks, I came to the conclusion that "art" must be one of the most interesting words in the English language.

The Arts of Old

If we go back to Latin roots, we find *ars, artis* meaning "skill." It is perhaps significant that the corresponding Greek word was τέχνη, the root of both "technology" and "technique."

Nowadays when someone speaks of "art" you probably think first of "fine arts" such as painting and sculpture, but before the twentieth century the word was generally used in quite a different sense. Since this older meaning of "art" still survives in many idioms, especially when we are contrasting art with science, I would like to spend the next few minutes talking about art in its classical sense.

In medieval times, the first universities were established to teach the seven so-called "liberal arts," namely grammar, rhetoric, logic, arithmetic, geometry, music, and astronomy. Note that this is quite different from the curriculum of today's liberal arts colleges, and that at least three of the original seven liberal arts are important components of computer science. At that time, an "art" meant something devised by man's intellect, as opposed to activities derived from nature or instinct; "liberal" arts were liberated or free, in contrast to manual arts such as plowing (cf. [6]). During the middle ages the word "art" by itself usually meant logic [4], which usually meant the study of syllogisms.

Science vs. Art

The word "science" seems to have been used for many years in about the same sense as "art"; for example, people spoke also of the seven liberal sciences, which were the same as the seven liberal arts [1]. Duns Scotus in the thirteenth century called logic "the Science of Sciences, and the Art of Arts" (cf. [12, p. 34f]). As civilization and learning developed, the words took on more and more independent meanings, "science" being used to stand for knowledge, and "art" for the application of knowledge. Thus, the science of astronomy was the basis for the art of navigation. The situation was almost exactly like the way in which we now distinguish between "science" and "engineering."

Many authors wrote about the relationship between art and science in the nineteenth century, and I believe the best discussion was given by John Stuart Mill. He said the following things, among others, in 1843 [28]:

> Several sciences are often necessary to form the groundwork of a single art. Such is the complication of human affairs, that to enable one thing to be *done*, it is often requisite to *know* the nature and properties of many things.... Art in general consists of the truths of Science, arranged in the most convenient order for practice, instead of the order which is the most convenient for thought. Science groups and arranges its truths so as to enable us to take in at one view as much as possible of the general order of the universe. Art...brings together from parts of the field of science most remote from one another, the truths relating to the production of the different and heterogeneous conditions necessary to each effect which the exigencies of practical life require.

As I was looking up these things about the meanings of "art," I found that authors have been calling for a transition from art to science for at least two centuries. For example, the preface to a textbook on mineralogy, written in 1784, said the following [17]: "Previous to the year 1780, mineralogy, though tolerably understood by many as an Art, could scarce be deemed a Science."

According to most dictionaries "science" means knowledge that has been logically arranged and systematized in the form of general "laws." The advantage of science is that it saves us from the need to think things through in each individual case; we can turn our thoughts to higher-level concepts. As John Ruskin wrote in 1853 [32]: "The work of science is to substitute facts for appearances, and demonstrations for impressions."

It seems to me that if the authors I studied were writing today, they would agree with the following characterization: Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. Since the notion of an algorithm or a computer program provides us with an extremely useful test for the depth of our knowledge about any given subject, the process of going from an art to a science means that we learn how to automate something.

Artificial intelligence has been making significant progress, yet there is a huge gap between what computers can do in the foreseeable future and what ordinary people can do. The mysterious insights that people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an art.

From this standpoint it is certainly desirable to make computer programming a science, and we have indeed come a long way in the 15 years since the publication of the remarks I quoted at the beginning of this talk. Fifteen years ago computer programming was so badly understood that hardly anyone even *thought* about proving programs correct; we just fiddled with a program until we "knew" it worked. At that time we didn't even know how to express the *concept* that a

program was correct, in any rigorous way. It is only in recent years that we have been learning about the processes of abstraction by which programs are written and understood; and this new knowledge about programming is currently producing great payoffs in practice, even though few programs are actually proved correct with complete rigor, since we are beginning to understand the principles of program structure. The point is that when we write programs today, we know that we could in principle construct formal proofs of their correctness if we really wanted to, now that we understand how such proofs are formulated. This scientific basis is resulting in programs that are significantly more reliable than those we wrote in former days when intuition was the only basis of correctness.

The field of "automatic programming" is one of the major areas of artificial intelligence research today. Its proponents would love to be able to give a lecture entitled "Computer Programming as an Artifact" (meaning that programming has become merely a relic of bygone days), because their aim is to create machines that write programs better than we can, given only the problem specification. Personally I don't think such a goal will ever be completely attained, but I do think that their research is extremely important, because everything we learn about programming helps us to improve our own artistry. In this sense we should continually be striving to transform *every* art into a science: in the process, we advance the art.

I can't resist telling another story relating science and art. Several years ago when I visited the University of Chicago, I noticed two signs as I entered one of the buildings. One of them said "Information Science," and it had an arrow pointing to the right; the other said "Information," and its arrow pointed to the left. In other words, it was one way for the Science, but the other way for the Art of Information.

Science and Art

Our discussion indicates that computer programming is by now *both* a science and an art, and that the two aspects nicely complement each other. Apparently most authors who examine such a question come to this same conclusion, that their subject is both a science and an art, whatever their subject is (cf. [25]). I found a book about elementary photography, written in 1893, which stated that "the development of the photographic image is both an art and a science" [13]. In fact, when I first picked up a dictionary in order to study the words "art" and "science," I happened to glance at the editor's preface, which began by saying, "The making of a dictionary is both a science and an art." The editor of Funk & Wagnall's dictionary [27] observed that the painstaking accumulation and classification of data about words has a scientific character, while a well-chosen phrasing of definitions demands the ability to write with economy and precision: "The science without the art is likely to be ineffective; the art without the science is certain to be inaccurate."

When preparing this talk I looked through the card catalog at Stanford library to see how other people have been using the words "art" and "science" in the titles of their books. This turned out to be quite interesting.

For example, I found two books entitled *The Art of Playing the Piano* [5, 15], and others called *The Science of Pianoforte Technique* [10], *The Science of Pianoforte Practice* [30]. There is also a book called *The Art of Piano Playing: A Scientific Approach* [22].

Then I found a nice little book entitled *The Gentle Art of Mathematics* [31], which made me somewhat sad that I can't honestly describe computer programming as a "gentle art."

I had known for several years about a book called *The Art of Computation*, published in San Francisco, 1879, by a man named C. Frusher Howard [14]. This was a book on practical business arithmetic that had sold over 400,000 copies in various editions by 1890. I was amused to read the preface, since it shows that Howard's philosophy and the intent of his title were quite different from mine; he wrote: "A knowledge of the Science of Number is of minor importance; skill in the Art of Reckoning is absolutely indispensible."

Several books mention both science and art in their titles, notably *The Science of Being and Art of Living* by Maharishi Mahesh Yogi [24]. There is also a book called *The Art of Scientific Discovery* [11], which analyzes how some of the great discoveries of science were made.

So much for the word "art" in its classical meaning. Actually when I chose the title of my books, I wasn't thinking primarily of art in this sense, I was thinking more of its current connotations. Probably the most interesting book which turned up in my search was a fairly recent work by Robert E. Mueller called *The Science of Art* [29]. Of all the books I've mentioned, Mueller's comes closest to expressing what I want to make the central theme of my talk today, in terms of real artistry as we now understand the term. He observes: "It was once thought that the imaginative outlook of the artist was death for the scientist. And the logic of science seemed to spell doom to all possible artistic flights of fancy." He goes on to explore the advantages which actually do result from a synthesis of science and art.

A scientific approach is generally characterized by the words logical, systematic, impersonal, calm, rational, while an artistic approach is characterized by the words aesthetic, creative, humanitarian, anxious, irrational. It seems to me that both of these apparently contradictory approaches have great value with respect to computer programming.

Emma Lehmer wrote in 1956 that she had found coding to be "an exacting science as well as an intriguing art" [23]. H. S. M. Coxeter remarked in 1957 that he sometimes felt "more like an artist than a scientist" [7]. This was at the time C. P. Snow was beginning to voice his alarm at the growing polarization between "two cultures" of educated people [34, 35]. He pointed out that we need to combine scientific and artistic values if we are to make real progress.

Works of Art

When I'm sitting in an audience listening to a long lecture, my attention usually starts to wane at about this point in the hour. So I wonder, are you getting a little tired of my harangue about "science" and "art"? I really hope that you'll be able to listen carefully to the rest of this, anyway, because now comes the part about which I feel most deeply.

When I speak about computer programming as an art, I am thinking primarily of it as an art *form*, in an aesthetic sense. The chief goal of my work as educator and author is to help people learn how to write *beautiful programs*. It is for this reason I was especially pleased to learn recently [33] that my books actually appear in the Fine Arts Library at Cornell University. (However, the three volumes apparently sit there neatly on the shelf, without being used, so I'm afraid the librarians may have made a mistake by interpreting my title literally.)

My feeling is that when we prepare a program, it can be like composing poetry or music; as Andrei Ershov has said [9], programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules.

Furthermore when we read other people's programs, we can recognize some of them as genuine works of art. I can still remember the great thrill it was for me to read the listing of Stan Poley's SOAP II assembly program in 1958; you probably think I'm crazy, and styles have certainly changed greatly since then, but at the time it meant a great deal to me to see how elegant a system program could be, especially by comparison with the heavy-handed coding found in other listings I had been studying at the same time. The possibility of writing beautiful programs, even in assembly language, is what got me hooked on programming in the first place.

Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write *grand* programs, *noble* programs, truly *magnificent* ones!

I discussed this recently with Michael Fischer, who suggested that computer programmers should begin to sell their original programs, as works of art, to collectors. The ACM could set up a panel to certify the authenticity of each genuinely new piece of code; then discriminating dealers and a new class of professionals called program critics would establish appropriate market values. This would be a nice way to raise our salaries if we could get it started.

Taste and Style

In a more serious vein, I'm glad that the idea of *style* in programming is now coming to the forefront at last, and I hope that most of you have seen the excellent little book on *Elements of Programming Style* by Kernighan and Plauger [16]. In this connection it is most important for us all to remember that there is no one "best" style; everybody has

his own preferences, and it is a mistake to try to force people into an unnatural mold. We often hear the saying, "I don't know anything about art, but I know what I like." The important thing is that you really *like* the style you are using; it should be the best way you prefer to express yourself.

Edsger Dijkstra stressed this point in the preface to his *Short Introduction to the Art of Programming* [8]:

> It is my purpose to transmit the importance of good taste and style in programming, [but] the specific elements of style presented serve only to illustrate what benefits can be derived from "style" in general. In this respect I feel akin to the teacher of composition at a conservatory: He does not teach his pupils how to compose a particular symphony, he must help his pupils to find their own style and must explain to them what is implied by this. (It has been this analogy that made me talk about "The Art of Programming.")

Now we must ask ourselves, What is good style, and what is bad style? We should not be too rigid about this in judging other people's work. The early nineteenth-century philosopher Jeremy Bentham put it this way [3, Bk. 3, Ch. 1]:

> Judges of elegance and taste consider themselves as benefactors to the human race, whilst they are really only the interrupters of their pleasure.... There is no taste which deserves the epithet *good,* unless it be the taste for such employments which, to the pleasure actually produced by them, conjoin some contingent or future utility: there is no taste which deserves to be characterized as bad, unless it be a taste for some occupation which has a mischievous tendency.

When we apply our own prejudices to "reform" someone else's taste, we may be unconsciously denying him some entirely legitimate pleasure. That's why I don't condemn a lot of things programmers do, even though I would never enjoy doing them myself. The important thing is that they are creating something *they* feel is beautiful.

In the passage I just quoted, Bentham does give us some advice about certain principles of aesthetics which are better than others, namely the "utility" of the result. We have some freedom in setting up our personal standards of beauty, but it is especially nice when the things we regard as beautiful are also regarded by other people as useful. I must confess that I really enjoy writing computer programs; and I especially enjoy writing programs which do the greatest good, in some sense.

There are many senses in which a program can be "good," of course. In the first place, it's especially good to have a program that works correctly. Secondly it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language.

Another important way for a production program to be good is for it to interact gracefully with its users, especially when recovering from human errors in the input data. It's a real art to compose meaningful error messages or to design flexible input formats which are not error-prone.

Another important aspect of program quality is the efficiency with which the computer's resources are actually being used. I am sorry to say that many people nowadays are condemning program efficiency, telling us that it is in bad taste. The reason for this is that we are now experiencing a reaction from the time when efficiency was the only reputable criterion of goodness, and programmers in the past have tended to be so preoccupied with efficiency that they have produced needlessly complicated code; the result of this unnecessary complexity has been that net efficiency has gone down, due to difficulties of debugging and maintenance.

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

We shouldn't be penny wise and pound foolish, nor should we always think of efficiency in terms of so many percent gained or lost in total running time or space. When we buy a car, many of us are almost oblivious to a difference of $50 or $100 in its price, while we might make a special trip to a particular store in order to buy a 50¢ item for only 25¢. My point is that there is a time and place for efficiency; I have discussed its proper role in my paper on structured programming, which appears in the current issue of *Computing Surveys* [21].

Less Facilities: More Enjoyment

One rather curious thing I've noticed about aesthetic satisfaction is that our pleasure is significantly enhanced when we accomplish something with limited tools. For example, the program of which I personally am most pleased and proud is a compiler I once wrote for a primitive minicomputer which had only 4096 words of memory, 16 bits per word. It makes a person feel like a real virtuoso to achieve something under such severe restrictions.

A similar phenomenon occurs in many other contexts. For example, people often seem to fall in love with their Volkswagens but rarely with their Lincoln Continentals (which presumably run much better). When I learned programming, it was a popular pastime to do as much as possible with programs that fit on only a single punched card. I suppose it's this same phenomenon that makes APL enthusiasts relish their "one-liners." When we teach programming nowadays, it is a curious fact that we rarely capture the heart of a student for computer science until he has taken a course which allows "hands on" experience with a minicomputer. The use of our large-scale machines with their fancy operating systems and languages doesn't really seem to engender any love for programming, at least not at first.

It's not obvious how to apply this principle to increase programmers' enjoyment of their work. Surely programmers would groan if their manager suddenly announced that the new machine will have only half

as much memory as the old. And I don't think anybody, even the most dedicated "programming artists," can be expected to welcome such a prospect, since nobody likes to lose facilities unnecessarily. Another example may help to clarify the situation: Film-makers strongly resisted the introduction of talking pictures in the 1920's because they were justly proud of the way they could convey words without sound. Similarly, a true programming artist might well resent the introduction of more powerful equipment; today's mass storage devices tend to spoil much of the beauty of our old tape sorting methods. But today's film-makers don't want to go back to silent films, not because they're lazy but because they know it is quite possible to make beautiful movies using the improved technology. The form of their art has changed, but there is still plenty of room for artistry.

How did they develop their skill? The best film-makers through the years usually seem to have learned their art in comparatively primitive circumstances, often in other countries with a limited movie industry. And in recent years the most important things we have been learning about programming seem to have originated with people who did not have access to very large computers. The moral of this story, it seems to me, is that we should make use of the idea of limited resources in our own education. We can all benefit by doing occasional "toy" programs, when artificial restrictions are set up, so that we are forced to push our abilities to the limit. We shouldn't live in the lap of luxury all the time, since that tends to make us lethargic. The art of tackling miniproblems with all our energy will sharpen our talents for the real problems, and the experience will help us to get more pleasure from our accomplishments on less restricted equipment.

In a similar vein, we shouldn't shy away from "art for art's sake"; we shouldn't feel guilty about programs that are just for fun. I once got a great kick out of writing a one-statement ALGOL program that invoked an innerproduct procedure in such an unusual way that it calculated the mth prime number, instead of an innerproduct [19]. Some years ago the students at Stanford were excited about finding the shortest FORTRAN program which prints itself out, in the sense that the program's output is identical to its own source text. The same problem was considered for many other languages. I don't think it was a waste of time for them to work on this; nor would Jeremy Bentham, whom I quoted earlier, deny the "utility" of such pastimes [3, Bk. 3, Ch. 1]. "On the contrary," he wrote, "there is nothing, the utility of which is more incontestable. To what shall the character of utility be ascribed, if not to that which is a source of pleasure?"

Providing Beautiful Tools

Another characteristic of modern art is its emphasis on creativity. It seems that many artists these days couldn't care less about creating beautiful things; only the novelty of an idea is important. I'm not recommending that computer programming should be like modern

art in this sense, but it does lead me to an observation that I think is important. Sometimes we are assigned to a programming task which is almost hopelessly dull, giving us no outlet whatsoever for any creativity; and at such times a person might well come to me and say, "So programming is beautiful? It's all very well for you to declaim that I should take pleasure in creating elegant and charming programs, but how am I supposed to make this mess into a work of art?"

Well, it's true, not all programming tasks are going to be fun. Consider the "trapped housewife," who has to clean off the same table every day: there's not room for creativity or artistry in every situation. But even in such cases, there is a way to make a big improvement: it is still a pleasure to do routine jobs if we have beautiful things to work with. For example, a person will really enjoy wiping off the dining room table, day after day, if it is a beautifully designed table made from some fine quality hardwood.

Sometimes we're called upon to *perform* a symphony, instead of to compose; and it's a pleasure to perform a really fine piece of music, although we are suppressing our freedom to the dictates of the composer. Sometimes a programmer is called upon to be more a craftsman than an artist; and a craftman's work is quite enjoyable when he has good tools and materials to work with.

Therefore I want to address my closing remarks to the system programmers and the machine designers who produce the systems that the rest of us must work with. *Please*, give us tools that are a pleasure to use, especially for our routine assignments, instead of providing something we have to fight with. Please, give us tools that encourage us to write better programs, by enhancing our pleasure when we do so.

It's very hard for me to convince college freshmen that programming is beautiful, when the first thing I have to tell them is how to punch "slash slash JOB equals so-and-so." Even job control languages can be designed so that they are a pleasure to use, instead of being strictly functional.

Computer hardware designers can make their machines much more pleasant to use, for example, by providing floating-point arithmetic which satisfies simple mathematical laws. The facilities presently available on most machines make the job of rigorous error analysis hopelessly difficult, but properly designed operations would encourage numerical analysts to provide better subroutines which have certified accuracy [cf. [20, p. 204]].

Let's consider also what software designers can do. One of the best ways to keep up the spirits of a system user is to provide routines that he can interact with. We shouldn't make systems too automatic, so that the action always goes on behind the scenes; we ought to give the programmer-user a chance to direct his creativity into useful channels. One thing all programmers have in common is that they enjoy working with machines; so let's keep them in the loop. Some tasks are best done by machine, while others are best done by human

insight; and a properly designed system will find the right balance. (I have been trying to avoid misdirected automation for many years, cf. [18].)

Program measurement tools make a good case in point. For years programmers have been unaware of how the real costs of computing are distributed in their programs. Experience indicates that nearly everybody has the wrong idea about the real bottlenecks in his programs; it is no wonder that attempts at efficiency go awry so often, when a programmer is never given a breakdown of costs according to the lines of code he has written. His job is something like that of a newly married couple who try to plan a balanced budget without knowing how much the individual items like food, shelter, and clothing will cost. All that we have been giving programmers is an optimizing compiler, which mysteriously does something to the programs it translates but which never explains what it does. Fortunately we are now finally seeing the appearance of systems which give the user credit for some intelligence; they automatically provide instrumentation of programs and appropriate feedback about the real costs. These experimental systems have been a huge success, because they produce measurable improvements, and especially because they are fun to use, so I am confident that it is only a matter of time before the use of such systems is standard operating procedure. My paper in *Computing Surveys* [21] discusses this further, and presents some ideas for other ways in which an appropriate interactive routine can enhance the satisfaction of user programmers.

Language designers also have an obligation to provide languages that encourage good style, since we all know that style is strongly influenced by the language in which it is expressed. The present surge of interest in structured programming has revealed that none of our existing languages is really ideal for dealing with program and data structure, nor is it clear what an ideal language should be. Therefore I look forward to many careful experiments in language design during the next few years.

Summary

To summarize: We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the *state of the Art*.

Note: The second paragraph on page 5 ("I can't resist...."), the fifth paragraph on page 7 ("I discussed this recently...."), and the first paragraph on page 11 ("Sometimes we're called upon....") were included in the lecture given in San Diego, but were added too late to appear in the originally published version.

References

1. Bailey, Nathan. *The Universal Etymological English Dictionary.* T. Cos, London, 1727. See "Art," "Liberal," and "Science."
2. Bauer, Walter F., Juncosa, Mario L., and Perlis, Alan J. ACM publication policies and plans. *J. ACM 6* (Apr. 1959), 121–122.
3. Bentham, Jeremy. *The Rationale of Reward.* Trans. from *Théorie des peines et des récompenses*, 1811, by Richard Smith, J. & H. L. Hunt, London, 1825.
4. *The Century Dictionary and Cyclopedia 1.* The Century Co., New York, 1889.
5. Clementi, Muzio. *The Art of Playing the Piano.* Trans. from *L'art de jouer le pianoforte* by Max Vogrich. Schirmer, New York, 1898.
6. Colvin, Sidney. "Art." *Encyclopaedia Britannica*, eds. 9, 11, 12, 13, 1875–1926.
7. Coxeter, H. S. M. Convocation address, Proc. 4th Canadian Math. Congress, 1957, pp. 8–10.
8. Dijkstra, Edsger W. EWD316: *A Short Introduction to the Art of Programming.* T. H. Eindhoven, The Netherlands, Aug. 1971.
9. Ershov, A. P. Aesthetics and the human factor in programming. *Comm. ACM 15* (July 1972), 501–505.
10. Fielden, Thomas. *The Science of Pianoforte Technique.* Macmillan, London, 1927.
11. Gore, George. *The Art of Scientific Discovery.* Longmans, Green, London, 1878.
12. Hamilton, William. *Lectures on Logic 1.* Wm. Blackwood, Edinburgh, 1874.
13. Hodges, John A. Elementary Photography: *The "Amateur Photographer" Library 7.* London, 1893. Sixth ed., revised and enlarged, 1907, p. 58.
14. Howard, C. Frusher. Howard's *Art of Computation* and golden rule for equation of payments for schools, business colleges and self-culture .... C. F. Howard, San Francisco, 1879.
15. Hummel, J. N. *The Art of Playing the Piano Forte.* Boosey, London, 1827.
16. Kernighan B. W., and Plauger, P. J. *The Elements of Programming Style.* McGraw-Hill, New York, 1974.
17. Kirwan, Richard. *Elements of Mineralogy.* Elmsly, London, 1784.
18. Knuth, Donald E. Minimizing drum latency time. *J. ACM 8* (Apr. 1961), 119–150.
19. Knuth, Donald E., and Merner, J. N. ALGOL 60 confidential. *Comm. ACM 4* (June 1961), 268–272.
20. Knuth, Donald E. Seminumerical Algorithms: *The Art of Computer Programming 2.* Addison-Wesley, Reading, Mass., 1969.
21. Knuth, Donald E. Structured programming with **go to** statements. *Computing Surveys 6* (Dec. 1974), 261–301.
22. Kochevitsky, George. *The Art of Piano Playing: A Scientific Approach.* Summy-Birchard, Evanston, Ill., 1967.
23. Lehmer, Emma. Number theory on the SWAC. *Proc. Symp. Applied Math. 6*, Amer. Math. Soc. (1956), 103–108.
24. Mahesh Yogi, Maharishi. *The Science of Being and Art of Living.* Allen & Unwin, London, 1963.
25. Malevinsky, Moses L. *The Science of Playwriting.* Brentano's, New York, 1925.

26. Manna, Zohar, and Pnueli, Amir. Formalization of properties of functional programs. *J. ACM 17* (July 1970), 555–569.
27. Marckwardt, Albert H. Preface to *Funk and Wagnall's Standard College Dictionary.* Harcourt, Brace & World, New York, 1963, vii.
28. Mill, John Stuart. *A System of Logic, Ratiocinative and Inductive.* London, 1843. The quotations are from the introduction, §2, and from Book 6, Chap. 11 (12 in later editions), §5.
29. Mueller, Robert E. *The Science of Art.* John Day, New York, 1967.
30. Parsons, Albert Ross. *The Science of Pianoforte Practice.* Schirmer, New York, 1886.
31. Pedoe, Daniel. *The Gentle Art of Mathematics.* English U. Press, London, 1953.
32. Ruskin, John. *The Stones of Venice 3.* London, 1853.
33. Salton, G. A. Personal communication, June 21, 1974.
34. Snow, C. P. The two cultures. *The New Statesman and Nation 52* (Oct. 6, 1956), 413–414.
35. Snow, C. P. *The Two Cultures: and a Second Look.* Cambridge University Press, 1964.

Categories and Subject Descriptors:

D.1.2 [**Software**]: Programming Techniques—*automatic programming*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management; K.7.0 [**Computing Milieux**]: The Computing Profession—*general*

General Terms:

Performance, Standards, Theory