PROGRAMMING: From Babbage to Backus

B. Randell

Abstract

This paper, prepared for the Babbage Bi-Centennial Conference on "Computing", at the Imperial College of Science, Technology and Medicine (London 1-3 July, 1991), gives a somewhat personal view of some of the origins and evolution of programming. It discusses Babbage's own contributions to what we now know as "programming", set against a background of the technology of his day, and the subsequent steps towards programming as developed for the early stored program electronic computers. Brief comments are also made on the later development of the subject, and on its present state and likely future.
Bibliographical details

RANDELL, Brian

PROGRAMMING: From Babbage to Backus. [By] B. Randell

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1991.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 327)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 327

Abstract

This paper, prepared for the Babbage Bi-Centennial Conference on “Computing”, at the Imperial College of Science, Technology and Medicine (London 1-3 July, 1991), gives a somewhat personal view of some of the origins and evolution of programming. It discusses Babbage’s own contributions to what we now know as “programming”, set against a background of the technology of his day, and the subsequent steps towards programming as developed for the early stored program electronic computers. Brief comments are also made on the later development of the subject, and on its present state and likely future.

About the author

Professor Randell has been a professor of computing science at the Computing Laboratory of the University of Newcastle upon Tyne since 1969.

Suggested keywords

BABBAGE, Charles
BACKUS, John
FORTRAN
HISTORY OF PROGRAMMING
SEQUENCE CONTROL
SOFTWARE ENGINEERING
STORED PROGRAM CONCEPT

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.642509
U.D.C. 519.68 (091)
PROGRAMMING:  
From Babbage to Backus

Brian Randell

Abstract

This paper, prepared for the Babbage Bi-Centennial Conference on "Computing", at the Imperial College of Science, Technology and Medicine (London 1-3 July, 1991), gives a somewhat personal view of some of the origins and evolution of programming. It discusses Babbage's own contributions to what we now know as "programming", set against a background of the technology of his day, and the subsequent steps towards programming as developed for the early stored program electronic computers. Brief comments are also made on the later development of the subject, and on its present state and likely future.

Keywords

Charles Babbage, John Backus, Stored Program Concept, Sequence Control, History of Programming, FORTRAN, Software Engineering.

1. INTRODUCTION

My aim in this paper is to give a somewhat personal view of some of the origins and evolution of programming; it is certainly not to attempt to provide a complete history of the subject. Indeed, given the difficulty of assessing the ways and extent to which any given achievement affected later developments, my account will be more in the nature of a partial chronology than a history. However, I will try to avoid the standard pitfall of a chronology - that of becoming mainly a catalogue of claimed "firsts". Such identifications are often misleading and controversial; in any case, with enough qualifications, almost anything can be so categorized.

Another pitfall I wish to avoid is that of giving just a "Whig interpretation" of history. Quoting [Cohen 1988]: "Whig historians produced chronicles of the heroes of the past, whose achievements were celebrated because they did well on a scale of values determined by the degree of accord with a present state of scientific knowledge and belief. . . . Historians of science since the 1950s have generally abandoned [this approach to history because] they have come to see the advantages of studying the
scientific thought of the past in the direct terms of the problems and intellectual currents of the time under which any work was done, rather than merely 'grading' it in a schoolmasterish way in terms of its degree of accord with the present.

Thus, though I have no pretensions in this brief account of advancing the state of historical studies into the origins of programming, I have aimed to provide at least some brief explanations of the nature and extent of the intellectual and technical achievements that were involved in a few selected developments. Moreover, it is important to realize that I have selected many of these particular developments more because I find them interesting from our current perspective than because of any contemporary importance or subsequent influence that I might believe or hope they have had.

One final disclaimer: the subtitle of this lecture has, I must confess, been chosen somewhat for its alliterative quality and should not be taken too literally. Indeed, my intent is to go further back even than the work of Charles Babbage, in describing precursors to what we now term "programming", and I am using John Backus' name mainly as representative of the group of language designers whose work I will refer to briefly at the end of this lecture.

2. THE PRE-BABBAGE PERIOD

One of the difficulties of discussing the historical origins of a subject is to know where to begin. Charles Babbage's ideas relating to what we now know as programming significantly surpassed what had gone before, and like virtually all of his work on computing were hardly to be matched, leave alone surpassed, for a century afterwards. Thus, regardless of whether or not they significantly influenced the modern development of the subject, they could make a very appropriate starting point for my account. However, I will start my story with a much earlier era, since the great degree of innovativeness Babbage demonstrated cannot be adequately appreciated, without some knowledge of the state of the "relevant arts" when he started his work, especially from an era in which computers have become totally ubiquitous.

A most important art in this regard was that of means for specifying a sequence of choices amongst a set of possible machine actions in such a way that the machine can carry out the sequence completely automatically. There were of course other specialized arts that Babbage needed for his Analytical Engine (quite apart from such general facilities as tools for the accurate machining of mechanical components). These included means of storing large quantities of retrievable and changeable numerical and logical data, and of performing arithmetic operations mechanically. And in fact technologies for these latter arts were nothing like as well established by the 1830s, as were pegged cylinders and Jacquard cards, the two technologies that Babbage planned to use as means for automatic sequence control for his Analytical Engine. But it is just these two technologies that I will take as my initial point of departure, for an account which of necessity will confine itself closely just to programming-related issues.

The pegged cylinder, still used in music boxes on sale today, though not I suspect for much longer, can (at least with hindsight), be traced back to the time of
Heron of Alexandria. In about 100 AD he described mechanisms involving the winding of a rope to and fro over the surface of a cylinder, from peg to peg, in such a way that when the cylinder was turned the rope wound and unwound irregularly, causing various other devices to perform a small sequence of actions. By this means several apparently miraculous effects were achieved, such as temple doors that apparently opened themselves and religious effigies that moved uncannily.

The technique of using the pegs themselves directly, rather than such a rope, as the means of causing other mechanisms to go through complicated series of repetitive motions can be traced back as far as the thirteenth century. For example, Arabic drawings dating from this era show mechanisms such as one used in a model boat, powered by water pressure from a tank above the deck, which turned a pegged cylinder so causing little model figures to row the boat around a pond, and others to bang drums and cymbals. (The written descriptions accompanying these drawings are so precise and detailed that there can be little doubt that the device was actually built and used - apparently for entertainment purposes at royal drinking parties!)

It is also known that similar pegged cylinder mechanisms were in use in Europe a century or so later century to control the movements of model figures decorating large church clocks, and the playing of their bells. In most cases each set of pegs around a given circumference of the cylinder simply controlled the occurrences of a given action, such as a particular movement of a marionette, or the sounding of a particular musical note - thus we can view each potential peg position as storing a binary digit. (In some other, later, devices the actual shape or length of the peg was significant, so more information was provided by each peg.) In all cases, however, until quite late on, it would seem that such pegged cylinders were regarded as an integral part of the machine they were controlling. The insertion and removal of individual pegs was sometimes facilitated, but the cylinder itself could not be readily replaced by another one in order to cause, say, a different tune to be played.

The idea of controlling a machine by sequencing information held on some clearly separate medium, using which a variety of different sequences could be prepared away from the machine, instead seems to have arisen first in the weaving industry, in fact in the early eighteenth century. (Interchangeable pegged cylinders, and for that matter other interchangeable media such as punched tapes and disks, were not used in automatic musical instruments and other automata until over a century later.)

However, through the efforts starting in the early 1700s of a small series of French inventors, automatic sequencing was applied to silk-weaving so as to produce figured silken cloth. Such sequencing devices finally became commercially successful in the first decade of the nineteenth century, when Jacquard devised a fully automatic draw-loom which used strung-together punched cards (acting essentially as a wide punched paper tape). Each card controlled the selection of warp threads that were to be raised ready for a single passage across the loom of the shuttle carrying the weft thread. The result was to cause a complex pattern to be woven into the cloth. The technology spread rapidly - so rapidly in fact as to cause considerable industrial unrest - and
thousands of examples of the Jacquard loom, as it came to be known, were in operation by the 1830s, including many in Britain.

From our point of view, Jacquard looms are also of interest as marking the first time that automatic sequence control was used for serious commercial purposes, as opposed to being used for "merely" impressing or entertaining, or even frightening, people. However this view of their relative importance, and hence implication concerning how they were appreciated at the time, owes overmuch to hindsight and twentieth century values. Indeed, it seems clear that for centuries what many people, even serious philosophers, found most fascinating was the idea that human-like mechanical figures could be made to perform life-like movements completely automatically.

For example, in the mid-18th century Vaucanson seems to have been much more famous for constructing and exhibiting such mechanical automata than for the very significant contributions he made to what later became known as the Jacquard technology. (He incidentally provides the first known link between the two technologies.) And Babbage himself commented on the fact that visitors to his famous soirées tended to be more impressed by his demonstrations of the mechanical automaton that he had acquired, which he called his "silver lady", than by his work on machines which were intended to be used for automating the production and printing various practically-useful mathematical tables.

Jacquard technology enabled much longer action sequences to be specified than did pegged cylinders, and hence could be used to control the weaving of extremely complex patterns - one famous early example being a woven silk portrait of Jacquard, fine enough to be taken for a steel engraving, whose weaving involved no less than 20,000 cards. (Babbage is known to have been the proud possessor of one of these portraits.) Moreover the sets of cards were manifestly independent of the machine they controlled, and the need for their production, on a grand scale, gave rise to a whole range of skills and tools.

In fact, designs were normally drawn out on squared paper, from the successive lines of which the holes to be punched in a series of cards could be determined directly. Machines were soon introduced to aid the correct punching of cards from such designs, and also the making of multiple copies of a given card, and of a duplicate of a card sequence. (A splendid set of such machines is to be found, along with historic looms by Vaucanson, Jacquard and others, in the Musée des Techniques of the Conservatoire des Arts et Métiers, in Paris.)

However, so direct was the step of transcribing a drawing in order to produce a card sequence that it would be misleading to regard Jacquard cards and their preparation as involving any real form of programming. Card strings could be tied into a simple single loop, so as to control the weaving of a repeated pattern. But each of the set of possible actions that could be requested always had exactly the same effect, and thus each card caused the machine to do exactly the same thing each time it was read; and the sequence of such actions was totally immutable.
It was in fact to be Babbage who, in connection with his plans for using both punched cards and pegged cylinders, was for the first time to get past, indeed far past, these restrictions. As a result he got very near, if not quite, to the ideas of (machine-level) programming and microprogramming as we now understand them.

3. BABBAGE AND HIS SUCCESSORS

Charles Babbage first became interested in automatic calculation in 1821, when he started his work on difference engines. However his ideas soon progressed far beyond that of a special-purpose calculating machine - in fact almost as soon as he started work on his first full size Difference Engine he became dissatisfied with its limitations. In particular, he wished to avoid the need to have the highest order of difference constant, in order to be able to use the machine directly for transcendental as well as algebraic functions.

In 1834 Babbage started active work on these matters, and on problems such as division and the need to speed up the part of the addition mechanism which dealt with the assimilation of carry digits. He developed several very ingenious methods of carry assimilation, but the time savings so obtained would have been at the cost of a considerable amount of complex machinery. This led Babbage to realize the advantages of having a single centralized arithmetic mechanism, the "mill", separate from the "figure axes", i.e. columns of geared wheels which acted merely as storage locations rather than as accumulators, each with their own adding mechanisms, as in his difference engines.

The complexity of his new "Analytical Engine" and in particular its mill, was such that Babbage sought a method of simplifying and making explicit the required sequencing of the activities of the various component mechanisms, for example in carrying out each individual addition, multiplication, etc. He made what was perhaps at the time a fairly natural choice, namely a pegged cylinder, for this sequencing requirement.

The full sophistication of Babbage's designs has only really become clear in recent years through Alan Bromley's detailed studies of Babbage's drawings and notations[Bromley 1982; Bromley 1987]. These studies have revealed that Babbage had fully detailed, and essentially workable, designs for mechanizing various highly complex algorithms. These algorithms were to be executed under the control of a pegged cylinder that Babbage envisaged as providing 100 or more pegs, or "studs", in each of 50-100 "verticals", i.e. lines of stud positions in a vertical line parallel to the axis of the barrel. Thus, in modern terminology, the barrel acted as a microprogram store with 50-100 words, each of 100 or more bits. (The use of modern terminology in describing historical devices is often rather misleading, but in this case seems fully justified, so great are the conceptual similarities involved.)

Complex sequencing possibilities were allowed for by the fact that the barrel could be made to rotate a small number positions either or forward or backward, or alternatively retain the same position, after a given vertical had been acted upon. Moreover, such movements of the barrel could be made conditional on the current state
of the machine, and for example could depend on whether a particular arithmetic value had changed sign. To quote Bromley: "The whole concept of a conditional sequence of actions in a machine, and in particular of a conditional dependence on the outcome of previous actions of the machine, is original to Babbage and to the design of the Analytical Engine. It is a concept of the most profound importance." (I will return to this point later.)

By such means, Babbage planned to control the execution of what were in many cases highly parallel algorithms. He worked out these algorithms with the aid of a number of different graphical notations, that he himself had invented, and which functioned effectively as what we would term timing diagrams, logic diagrams, state-transition diagrams, and micro-program walkthroughs. In fact the logical sophistication of these algorithms, and indeed of the overall design of the Engine and its "microprogramming", exceeded that of many of the first generation of electronic computers.

Babbage had very early on decided that his machine should be of wide utility. Initially he also planned to use a pegged cylinder (with removable pegs) also for controlling the sequence of major operations executed, and the choice of operands to be used, but very soon decided to use Jacquard cards instead - for what it seems very fair to describe as the programmed control of his machine. He took advantage of the fact that these cards were strung together, to plan on the provision of means for "backing-up" (i.e. reversing through) a controlled number of cards, so as to be able to have cycles of operations, and to provide for alternative sequences to be executed. In so doing, he developed a very full understanding of many of the important conceptual advances that were provided by his planned use of Jacquard cards (for what he termed "formulae").

Basing his claim on the unbounded number of cards that could be used to control the machine, the ease with which complicated conditional branches could be built from a sequence of simple ones, and the fact that automatic card input and output, and multiple precision arithmetic were to be provided, he stated that [Babbage 1837]:

"... it appears that the whole of the conditions which enable a finite machine to make calculations of unlimited extent are fulfilled in the Analytical Engine. ... I have converted the infinity of space, which was required by the conditions of the problem, into the infinity of time."

He found the concept of conditional branching particularly fascinating from a philosophical point of view. Indeed, in his book "The Ninth Bridgewater Treatise" (an unsolicited contribution to a series of theological texts) he devoted an extended section to a discussion of how a machine (or the Universe), if controlled by a program that used conditional branching, could exhibit surprising, even apparently miraculous, behaviour. Moreover in an intended introduction to his machine, unpublished in his lifetime, Babbage included a very perceptive discussion of the problems of designing efficient programs, and of what could be done to cope prevent or tolerate faults in the machinery of the Analytical Engine itself, in the input data provided for it, and in the programs that were used to control it.
Babbage did not, however, work out the details of his program control to the same level of detail as his microprogram control, and a number of his design decisions, and omissions, seem rather strange to our eyes. For example, he planned to use two separate strings of cards to control the Analytical Engine. The "operation cards" controlled the sequence of operations to be performed, the "variable cards" identified the storage locations which were to be used to provide their operands and to receive their results - and means were to be provided for the two strings to be moved forwards or backwards independently of each other.

As explained by Bromley [Bromley 1982] it is clear that Babbage had what at the time seemed good reasons for the separation he made between operation and variable cards. But the result was that he apparently never arrived at the idea of what we would recognize as instructions, each identifying both an operation and its operands. Similarly, his ideas on card sequencing, and on loop control, were not fully worked out - and to the best of our knowledge he never planned on providing means for the machine to calculate the address of a variable. Indeed, the various "formulae" he worked out, many of which were included in the annotated translation by Ada Lovelace of an Italian report on Babbage's lectures in Turin on his plans [Lovelace 1843], are all really annotated traces rather than what we would call programs.

Such comments should not be taken as "criticisms" of Babbage's work on programming. Rather, his achievements, and the way they range from detailed engineering design to deep understanding of the conceptual issues and consequences involved, are immensely impressive, especially given the level of knowledge and technology that existed at the time. One can argue about whether he was so far ahead of his time as to be pursuing unreasonable and unrealistic goals, and what impact his efforts and the publicity that surrounded them had subsequently. However what is clear is that it was, as far as we know, another seventy or so years before anyone else took a further step beyond Babbage's programming ideas towards those on which the modern computer is used.

This person was an Irish accountant, named Percy Ludgate. In 1903 at the age of twenty he started work on a novel method of performing decimal arithmetic by mechanical means, quite different from any incorporated in any of the fairly considerable variety of desk calculators that by then were on the market. So striking are the differences between Ludgate's plans and Babbage's that there seems little reason to dispute Ludgate's statement that he did not learn of Babbage's work until the later stages of his own. It does however seem likely that Babbage was the eventual inspiration for Ludgate to investigate the provision of a sequence control mechanism for his planned calculating machine.

The advance that Ludgate made was simply that of planning to use a single punched tape, containing both operation codes and decimal operand addresses, to control his machine. Control transfers would then just involve moving the tape the appropriate number of rows forwards or backwards. Moreover he also envisaged the use of what we would now call subroutines, represented by sequences of similarly-coded rows of perforations around the circumference of special cylinders. (One such
cylinder was to be provided for performing division, which he did not envisage providing as a built-in operation.) There is no evidence that he ever tried to construct his machine, which he apparently worked on alone, and in his spare time. Indeed all that is known of his work comes from the two papers he published [Ludgate 1909; Ludgate 1914], the second of which was a survey paper, containing a description of Babbage’s machine. This incidentally was apparently one of the two main sources of Howard Aiken’s later "discovery" of Babbage.

Perhaps just one other development that occurred within the century following Babbage’s invention of the Analytical engine is worth mentioning here - namely the work of Torres y Quevedo, a renowned Spanish scientist and engineer. Torres did much work on the development of electromechanical digital devices, and picked on Babbage’s Analytical Engine as an important and interesting challenge to demonstrate their power and utility. In 1914 he published a paper [Torres y Quevedo 1914] showing how the various components of an Analytical Engine might be built from his technology, something which he actually did some years later. This paper is of relevance to this account mainly because it contains a description of a scheme for floating point arithmetic. This form of arithmetic was later re-invented (several times) and, as will be discussed below, its provision in some of the early electronic computers had a significant effect on the ways in which they were programmed.

4. THE STORED PROGRAM CONCEPT

As viewed from our vantage point, perhaps the really crucial machine-level programming concept remaining to be devised was the "stored program concept". There has been much controversy over the credit due for this development, which I will not enter into here, some of which at least is due to lack of agreement as to just what the concept involves.

The machines I have mentioned so far were all controlled by a program held on some read-only medium, such as punched cards or tape, quite separate from the (of course alterable) storage device used to hold the information that was being manipulated by the machine. Nevertheless, Babbage at least was aware of the possibility, and the potential importance, of having the Analytical Engine be able to generate and output its own programs, i.e. punched card formulae. (This he viewed as a good means of reducing the incidence of errors in lengthy formulae.)

The advent of electronics, and the first attempts at building programmable electronic calculating devices in the late 1930s and early 1940s exposed a need for some means of representing programs

(i) which could cope with the required program sizes,

(ii) whose access speed matched that of the (fully-electronic) operations they were controlling, and

(iii) which allowed adequately fast means of replacing a program whose task had been finished with the next program to be executed.
This problem had not existed with mechanical or electro-mechanical devices, whose calculation speeds were reasonably well-matched to the speed with which the cards or tape that controlled them could be read. And the system of plugs and cables used for programming the ENIAC, though was well matched to the calculation speed of its electronics, was such that the task of replacing one program by another could take several days. (An essentially similar system used a few years earlier for the Colossus did not take so long, simply because the programs that could be set up were much shorter and simpler.) This situation led to the realization, probably first in the ENIAC/EDVAC team at the University of Pennsylvania, of the advantages of storing the program within the computer, in a memory that could be read at electronic speeds during program execution [Eckert 1945]. Then the fact that different types of application had greatly differing relative requirements for instruction and (both variable and constant) data storage soon led to a realization of the practical benefits of using a single store for all three types of information [Eckert 1947].

To some commentators this constitutes the set of ideas comprising the stored program concept. However to many, myself included, the concept also has strong connotations of the computer being able to construct, manipulate and then (surpassing the notion that Babbage had arrived at over a century earlier) execute its own programs, all completely automatically. With this latter view the stored program concept becomes an engineering approximation to the theoretical universal automaton that Turing had postulated in his (now) famous 1936 paper [Turing 1936] - i.e. a machine which is general-purpose in a very fundamental mathematical sense as well as in a very practical sense. Thus, given the practical requirement of replacing the Turing Machine's infinite tape by a sufficiently large store random access store, it is crucial for the computer to be able to calculate the addresses that are used to access the store, rather than only be able to use pre-calculated (i.e. fixed) addresses.

In fact by these standards the first (1945) design for EDVAC [von Neumann 1945] does not qualify as a stored program computer. Although data and instructions were to be held in the same store their representations were quite distinct, and no means were provided for converting data items into instructions. Furthermore, normal arithmetic operations could not be applied to instructions - though the address field of an instruction could be modified.

This inadequacy, as we would of course view it, was not present in Turing's proposed design for an Automatic Computing engine (ACE), which slightly post-dated the EDVAC report, and was also very quickly remedied in the EDVAC design. [Carpenter and Doran 1977] In both cases, however, what now seem very awkward techniques of program self-modification were needed in order to make the machine calculate the addresses of variables - since neither the idea of index registers (B-lines as they were to be called, at Manchester, where they were invented) or of indirection had yet arisen. However, once all these aspects of the stored program concept had been provided, although there was a huge space of possible instruction formats and sets still to explore and exploit, (machine-level) programming essentially as we know it now had arrived. Needless to say, the great importance of this "event" has only become evident with the benefit of hindsight.
5. PROGRAMMING THE EARLY ELECTRONIC COMPUTERS

Many of the earliest programming developments arose at Cambridge, which had in EDSAC the world’s first practical stored program electronic computer to go into more-or-less regular operation and provide a real computing service. The influence of these Cambridge developments, on both sides of the Atlantic, was considerably enhanced by the publication of “Wilkes, Wheeler and Gill” [Wilkes, Wheeler et al. 1951]. (It has been claimed [Knuth and Trabb Pardo 1976] that the three-volume Goldstine and von Neumann report from the IAS Project at Princeton [Goldstine and von Neumann 1947-1948] laid "the foundation for computer programming all over the world". However Campbell-Kelly [Campbell-Kelly 1982] argues very believably that this "would appear to overstate considerably the influence of the reports on programming in Britain").

In particular, Cambridge pioneered the use of a library of closed subroutines, of tracing and post-mortem techniques of debugging, and of a (simple but effective) partly-symbolic re-locating assembly program. However the earliest machines in both the USA and the UK were nearly all one of a kind, each with a more-or-less different instruction set. Thus many of the programming techniques, and all the actual programs, that were developed tended to be specific to a particular machine. (The early machines which derived from Turing’s work, namely the Pilot ACE and DEUCE, and in the USA the Bendix G-15, were perhaps more different than most. And they were programmed at a level more reminiscent of subsequent microprogrammable machines, and were at once more in need of, and more difficult to provide with, efficient symbolic languages than the more conventional machines.) Indeed, programming was typically still machine-dependent even when based on one of the early interpretive systems that provided a (relatively) high level language interface, (e.g. for performing matrix calculations) that completely hid the machine’s basic instruction set.

Much early work on high level languages was in fact based on the use of interpretation techniques, rather than what we now know as compilation, and tended to be motivated by the difficulties and costs of programming a given type of machine, rather than concerns about program portability. Luckily, since in the early machines floating point arithmetic had to be provided by subroutine, the overheads caused by the actual interpretation could often be largely ignored, at least in many scientific applications. But hardware developments proceeded rapidly, permitting the provision of built-in floating point (to the undisguised dismay of some system programmers) and so increasing the requirement for effective compilation techniques.

Particularly in the USA, such developments were also usually accompanied by the provision of what at the time seemed to British eyes to be huge memories. The result was that for some time much more ambitious so-called “automatic programming techniques” were attempted in the USA than in Europe, with John Backus’ FORTRAN project in IBM, which started in 1954, being the most impressive example of such work. This project’s particular significance was that it successfully demonstrated that it was possible to produce machine-level programs, from a high level language, that were in general reasonably competitive in performance with manually produced programs.
This achievement involved the development of highly original (and complex) optimization and register allocation techniques for a compiler which was distributed on the then extremely impressive number of 20,000 punched cards - shades of the woven Jacquard portrait!

During the 1950s digital computers also progressed from being largely one-of-a-kind developments, used mainly for scientific and engineering calculations by the institutions that had designed or built them. They became commercial products, manufactured in at least modest quantities by a growing number of computer manufacturers. Though still very large and expensive, they began to be put to a variety of commercial, industrial, and military and other government uses by many different organizations. In the scientific arena, their use rapidly overtook that of analogue computers, and made possible far more extensive computations than had ever been feasible even with large batteries of desk calculators, such as the first numerical weather predictions. In the commercial arena they became commonly employed as additions to, or replacements for, large punched card tabulating installations used for major data processing tasks. And it seems to have been in this arena that, influenced strongly by pressure from the US Department of Defense, the most effective early work was done on machine-independent programming languages - COBOL being the result.

Towards the end of the 1950s interest in the possibility of defining a machine-independent programming language for scientific programs arose, particularly in Europe where FORTRAN had yet to gain much of a foothold, and where many had a strong wish to combat IBM's growing market dominance. The result of course was ALGOL - whose importance we now can recognize as mainly resting on the influence it had on much of the future course of language design, rather than on the extent of its own use. (The elegance and generality of ALGOL would appear to refute the common wisdom about the results of "design by committee" - but such a comment by implication underestimates the great extent of the contribution made by Peter Naur through his editing of a set of committee decisions in order to produce the ALGOL 60 Report.) However for all the attention being paid to high-level languages, for a long time assembly language was to remain predominant, in practice into the 60s. And the move away from such languages that FORTRAN led probably constituted the largest single improvement in programming that we have yet seen.

7. THE 1960s AND THE "SOFTWARE CRISIS"

The 1960s saw computer centers set up in many medium and large-scale scientific, government and business establishments. Additionally, in the mid-1960s, with the advent of the first commercial minicomputer, Digital Equipment Corporation's PDP-8, computers started to appear in laboratories and similar environments. The first university computing science departments were created, usually as offspring of mathematics or electronic engineering departments, and set about trying to justify their choice of title.

Computing and communications technologies started to come together. For example, the first on-line airline seat reservations system, namely the SABRE system
built by IBM for American Airlines, and AT&T's first electronic switching system were introduced. The ARPANET project, to develop a resource sharing network, was launched by the US Defense Department's Advanced Research Projects Agency (ARPA). Integrated circuits, albeit only at the SSI (Small Scale Integration) level, began to become available early in the 1960s, spurred in part by the needs of the US aerospace industry. They largely replaced the use of discrete transistors, and were challenging the dominance of ferrite core memories by the end of the decade. These circuits enabled the development of computers that were vastly more powerful and reliable than their predecessors.

In those days, for all the gradually growing use of high-level languages, shifting to a new computer normally implied having to abandon or rewrite existing applications programs because of hardware incompatibilities. This situation led IBM in 1964 to introduce a range of compatible machines of varying power and capacity, the System/360 series, to replace their previous distinct types of mutually incompatible scientific and commercial computers. This strategy of having a range of compatible machines was soon followed by other manufacturers - in some cases by introducing actual System/360-compatible machines.

The term "software" came into use, though as yet systems software was usually provided "free" with the hardware by the manufacturer, and applications software was normally designed specially for particular clients and particular computers. It was perhaps only when, in 1969, IBM "unbundled" its software by pricing it separately from its hardware that software became a commodity. Memory capacities increased, and the first time-sharing systems were brought into use, starting with MIT's CTSS in 1963. They were largely motivated by a wish to improve programmers' and users' ability to interact with their computers, though batch-processing systems remained the more common.

Increasingly ambitious applications and systems software projects were being undertaken, and organizations found themselves becoming much more dependent on large and complex computer systems than had previously been the case. Although there were some major success stories, one result was a growing concern about software cost and software project schedule over-runs, and about failures, some quite spectacular, to achieve performance and reliability goals.

This was the background to the 1968 NATO Conference at Garmisch in Germany [Naur and Randell 1968] with the deliberately provocative title of "Software Engineering", which started the software engineering bandwagon rolling. One of the most notable aspects of this conference was the willingness of the participants, "about 50 experts from all areas concerned with software problems - computer manufacturers, universities, software houses, computer users, etc.", to admit the extent and gravity of these software problems. Quoting a comment made by Dijkstra at the Conference: "The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement".
Dijkstra and I have both since gone on record as to how the discussions at this conference on the "software crisis", and on the potential for software-induced catastrophes, strongly influenced our thinking and our subsequent research activities. In Dijkstra's case it led him into an immensely fruitful long-term study of the problems of producing high quality programs. In my own case, it led me to consider the then very novel, and still somewhat controversial, idea of design fault tolerance. Suffice it to say that our respective choices of research problem suitably reflected our respective skills at program design and verification.

8. POST GARMISCH

The early 1970s saw the introduction of the first microprocessors. The extremely limited facilities of the early microprocessors initially caused them to be treated more as complex electronic components than as real programmable computers. They were as a result regarded with more interest by electronic engineers than computer scientists at first - the results were setting the programming field back by years, some computer scientists claimed at the time - a point I will return shortly.

Computer networking developed rapidly. Wide area packet-switched networks were introduced by various organizations in a number of countries. At XEROX PARC the Ethernet local area network was developed, in connection with what turned out to be a very influential program of research on distributed systems, personal workstations and human-computer interaction. Ironically, it was to be other companies, including the personal computer manufacturers, who exploited the results of this research most successfully.

Against this background, continuing progress was made in the programming field, in terms of theoretical foundations, techniques, languages and tools - topics which will I am sure be treated fully by other speakers at this conference. I therefore do not propose to enter far into the perennial debate about the merits of rival programming languages, or to discuss programming techniques and tools, except to point out that the whole subject of programming seems to have bifurcated into two somewhat different topic areas.

One of these is concerned principally with the tasks of individual programmers, the other the problems that relate more to those of sizeable programming teams. The distinction between these issues was already recognized at Garmisch, though it is of course not always clear cut. In fact one thing I personally find attractive about object-oriented programming is its relevance to both issues. (I will defend myself against a charge of "trendiness" by bragging that already in 1968 I and the other signatories of the Minority Report on Algol68 were of the view that Simula67, the original source of the approach we now call "object-oriented", was a much more significant step forward in language design than Algol68.)

But I, with others, would question whether any of these developments, and in particular any of the many languages, have had as much beneficial impact on the programming process as the introduction of FORTRAN provided. (Admittedly this view is not shared by John Backus, who clearly would now prefer to be judged by his
subsequent work on functional programming than on his original contributions to FORTRAN.) I do not deny the value of a number of the programming languages that have been devised over the last twenty or so years - it is just that the step from assembly language to FORTRAN had much more impact than subsequent transitions from one high level language to another, even when the transitions are not just amongst different flavours of imperative languages, but also to and amongst declarative languages.

In fact I think one needs to look beyond the research and advanced development activities concerned directly with the programming process at the surroundings of the subject to see what has caused the most significant changes in the field of programming over the last decade. The external influence that I have in mind is the explosive growth of the personal computer market. This made it practical to apply computers ever more widely, and to involve ever more people in the design of applications and systems software. As a result, particularly in the personal computer field, there has been enough production of similar software tools and application programs for the normal processes of competition, and "survival of the fittest", to work fairly effectively.

Thus, despite my earlier comments about the first microprocessors, much of the best personal computer software is now of much higher quality, particularly with regard to usability, than most mainframe software. And with so many more people engaged in the creation of software, there is rapid development of many new and innovative applications, and of programming libraries and tools. It is these often rather pragmatic developments that I feel are having, at least at present, the largest impact on the way in which industrial programming is changing.

10. THE FUTURE

Just as one needs to assess past programming ideas and developments against their original backgrounds, one cannot usefully consider the future of programming in isolation from the industry that it forms part of. The computer industry as a whole has grown extremely rapidly, the principal factor in this growth being of course progress in first electronic and then microelectronic (and perhaps in the future opto-electronic) technologies, and also in various storage technologies. This progress has led to continued exponential growth in all sorts of measures such as main memory size, processor speed, storage access rate, etc., and has of course made the "personal computer" revolution possible. But the important thing is that these exponential growth rates can, with some confidence, be predicted to continue for a number of years to come.

The speed with which hardware developments are occurring is often, but very misleadingly, contrasted with the more modest but nevertheless significant rate of improvement that is occurring in software productivity. However it is hardware technology which is developing so dramatically. The problems of hardware logic design, and the rate of productivity improvement being achieved, are essentially similar to those in software. Indeed hardware and software design methods are converging, as well as improving as new construction methods and tools are developed, fueled as I
indicated above by the ever growing number of people involved, and as more adequate theoretical foundations and effective higher-level notations are created.

Taking all these various factors into account in predicting how the world of computers and their programming will change is not easy. In fact it is as hard to predict what the next forty years of computing will bring as it would have been to foresee the developments of the past forty years in 1950. It is one thing to estimate how processing speeds and costs will change, and perhaps how our ability to design and implement comparatively well-understood applications will improve. It is quite another to predict the effect that current research in software engineering and in programming methodology will have on the great mass of programming, leave alone what new, and perhaps revolutionary, programming tools will be thought up (e.g. the next decade's equivalent of the spreadsheet program). Let us just hope that the results are worthy of the man whose bi-centenary we are celebrating at this conference, and whose contributions to programming I have tried, however inadequately, to describe and assess in this paper.

REFERENCES


