Software—Historical perspectives and current trends

by WALTER F. BAUER and ARTHUR M. ROSENBERG

Informatics Inc.
Canoga Park, California

SUMMARY

The importance of the study of history is to understand the forces which produce the events. Thus possibilities and prospects for the future may be better understood and evaluated. The development of historical perspectives as opposed to a historical record seems to have advantages in achieving that goal. This is especially true of programming where the award of historical "firsts" seems to be difficult at best.

This paper traces the historical evolution of software in the context of the developing computer hardware technology. It has been roughly only twenty years in which concepts of software have evolved from very crude beginnings into a sophisticated element of the expanding computer industry. It is time to take stock of the past and start to focus on the practical directions that software, as a technology and as an industry, will take.

The genealogy of software is developed, starting from the early 1950's when the most elemental concepts were subroutines, simple assemblers and simple diagnostic programs. From these simple concepts, through a process of successive additions of complex structures, came compilers, complex operating systems, and on-line systems programming. A newer concept, the "software product," is discussed including its business implications. A related subject, the development of machine aids to programming, is also a topic of discussion.

Advanced compilers and software language apparently come from the lineage of subroutines and simple assemblers. Advanced operating systems seem to have more in common with early interpretive programs than with any other possible antecedent. It seems clear that modern time-sharing systems were affected greatly by early on-line systems accomplished as part of the first military systems and, before that, from early diagnostic programs for finding programming and machine errors.

The future will probably not see any more proliferation of universal languages. For example, in the rapidly growing business data processing world, COBOL will continue to be used with COBOL shorthand and COBOL generators increasing in importance. A language-compatible set of tools for debugging, maintenance and documentation will be emphasized. Applications-oriented tools will receive much more attention. The trend starting with early report generator techniques and continuing with the powerful file management system approach will flourish. Machine emulation and software transferability will become of paramount importance with the relative increase of software over hardware costs.

Although important progress has been made, it is a general indictment of the programming profession working in industry that there is such a great time gap between concept and practical reality of a new idea.

INTRODUCTION

Computer programming is a unique and modern professional activity. The end product of programming is software. Programming is the process of developing procedures for a computer. The procedure itself is a piece of software.*

The computer is unique. It is an instrument of great versatility and universality. In the history of technology it is difficult, even impossible, to cite a preceding development so universally applied. Programming, which makes the computer useful, is equally unique. The computer, programmed appropriately, is a universal and powerful device for handling information.

There is little in man's history that can be cited as a direct antecedent of programming. Computational pro-

* The term, software, is sometimes taken to mean all activities, processes, and procedures surrounding the modern computer; in this definition, it includes everything related to the computer except hardware.

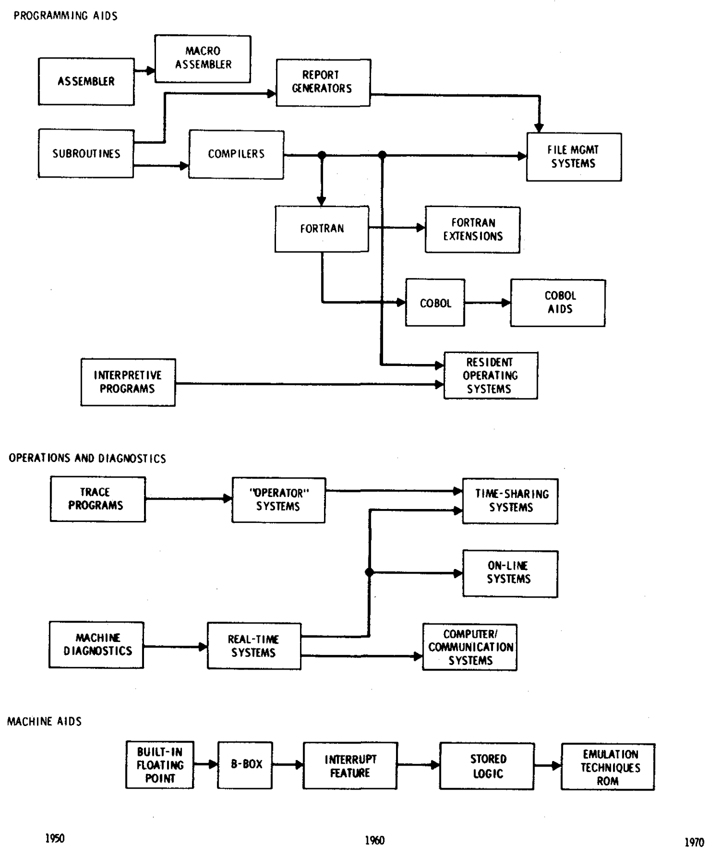

Figure 1—Early system geneology

cedures of all kinds, including those for desk calculators, offer some precedent, as do office procedures for business data. Further precedent is found in formalized rules for musical composition, first stated in modern terms by Bach in "The Well-Tempered Klavier," and in the setup procedures for such machines as the Jacquard loom. Precedent may also be sought in the rules of certain games which require playing by a procedure to win, such as tic-tac-toe, mill, and o-wah-ree. Such games extend back into human prehistory.

A good deal of the power and versatility of the stored program electronic computer is found in its ability to alter its future procedures on the basis of results of past procedures. This ability of the computer gives it that automatic character which permits it to carry out exceedingly involved and complex tasks, guided by interrelated program structures.[1]

The writing of a machine language program for the computer of typical design is a straightforward, but monumentally detailed and onerous task, especially if done with no aids. Programmers of the earliest electronic computers soon tired of such work and chafed under its limitations. They were led, quite naturally, to use the computer itself to make programming easier and more versatile. The development of programs on behalf of programming itself—to make programming less burdensome and expensive—got under way. These activities were called "automatic programming" in the middle of the 1950's.[2]

In the early 1950's, the process was referred to as the preparation of utility and service programs; the "automatic programming" label came later. Nowadays the practice is usually referred to as "systems programming." Whatever the name, the process is one of developing languages and procedures, and computer programs to operate in conjunction with these languages and procedures so that the computer itself can be the principal instrument to make programming easier.

EARLIEST CONCEPTS AND OBJECTIVES

The overall objective of systems programming is twofold: to make programming easier and to make the machine run more efficiently. Most of the programming sophistication of the early 1950's was aimed at programming efficiencies, although the great shortage of machine time, and its great cost, put heavy pressures on systems which used large amounts of machine time.*

Four areas of the programming task were targets for development of programming efficiencies. These were: the conceptual phase, consisting of architecture, design and specification; the programming or coding phase; the checkout phase; and finally, the modification and program maintenance phase. The earliest system software was aimed at making the coding job more efficient. A simple example is found in the replacement of the machine language operation code with an operation mnemonic.

In later years the targets for improved efficiencies turned to other ares such as design, program checkout and program modification. Thus, the Holy Grail of the programmer has been the development of systems to make programming efficient; but, the statement of the problem is more complex than that. For a given set of data processing functions and for a given degree of com-

* The question of objectives of system programming is an interesting philosophical point. In modern systems is the objective to make the computer run faster and keep programming tractable? Or is the objective the other way around—make programming easier and keep machine efficiencies reasonable? It would appear that a strong case can be made for the following: The 1960's, with increasingly complex applications and machines, systems programmers, mostly as a defensive measure, emphasized machine efficiencies to insure that programming became more efficient in complex environments. That notwithstanding, the authors believe unqualifiedly that in view of high labor costs and decreasing machine costs, the objective clearly should be the increasing of programming efficiencies.

plexity of the data processing problem, the programmer wishes to make his programming easier and to make the machine run faster. Success is elusive since new solutions must be found in an ever-increasingly complex environment and an ever-increasing set of data processing functions to be performed.*

SOFTWARE GENEALOGY

A historical perspective of how programming developed may give us better understanding of present and future developments in the list of the past. While citations of early milestones are given, no attempt is made here to record history per se, as others have done.[3] The concern is with the major events of programming development over the years, not with attributing invention or giving credit for milestones achieved. Typically, no matter what the programming technique and the alleged development time, it is possible to find someone who claims the same accomplishment at an earlier date.

There is often little unanimity of credit for earliest development. The time of conception of an idea is uncertain, and it is at least debatable whether the milestone is achieved upon conception or upon execution. To a considerable extent, programming structures are unclear, relations among programming methods cloudy, and programming elements themselves not always characterizable. Notwithstanding these difficulties of being able to distinguish, it is instructive to consider the main directions of programming development. One looks for the essence of what was developed, why it was developed, and, ultimately, what the future may bring.

The earliest of system programming activities seem to be divided among subroutines, interpretive routines, assemblers and loaders. To these may be added early attempts at diagnostics, both hardware and software. Machine aids to programming, that is, hardware design changes to make programming easier, have also come with some frequency beginning early in computer history. While this latter topic is not, strictly speaking, a programming development, it is related enough to be considered. It is as true today as it was twenty years ago: programming and machine design and characteristics are inextricably related; even today's "applications" programmer must know hardware characteristics. Thus, the three main channels of development can be characterized as the search for language power, the search for diagnostic power, and the search for machine aids to programming.

* "The problems of programming at this point appear as yet mostly unsolved. . . ." Report of the Discussion Group on Universal Codes. Proceedings of the First Joint AIEE-IRE Computer Conference, Philadelphia, 1951.

Figure 1 is a chart showing the genealogy of major software developments, emphasizing the earliest periods and how the earliest developments gave rise to later ones. While there is no claim made here for comprehensiveness or precision, the figure can serve in the purpose of discussing the structure and evolution of programming.

The beginnings of the development of programming structures and systems can be traced back to the emphasis of subroutines—their structure and use. The subroutine idea came about, naturally and quite obviously, by the programmer's observance that certain segments of the code were used or could be used repeatedly. From then on, ideas grew in proliferation. Macro instructions and pseudo instructions were developed as a convenience for calling subroutines and to specify the parameters to be used by the subroutine. One of the earliest technical writings on the subject of subroutine use and structures was that of Wilkes, Wheeler and Gill.[4] Needless to say, the development of the subroutine idea gave rise to the compilation of many subroutines into a running program and therefore to a major enhancement of simple assembly language programming, as well as to the compiler concept.

There seems little doubt that early programmers were led to interpretive programming as a simple extension of the subroutine idea. If, instead of occasional transfer of control to a subroutine and the interpretation of a macro or pseudo instruction, one considered that the entire running program consisted of macro instructions which are to be "interpreted" by the subroutines, a wide range of possibilities and a language of considerable power could be developed. Among the first interpretive programs were the ones for floating point arithmetic.

It is interesting to digress for a moment to discuss the similarities between the earliest subroutine structures and today's complex operating systems. In the earliest square root subroutine, for example, one parameter was supplied; namely, the number from which the square root was to be extracted. Today's operating systems are complex structures consisting of many subroutines. During the course of using the operating system, many parameters are supplied to the operating system for correct action: the programmer supplies the file from which information is to be extracted; he supplies the part of the program to be activated in the case of an interrupt on a certain channel; and, he supplies the output device as well as the form of the output desired.

This then characterizes programming and the development of programming systems: simple structures are used in multiplicity to create complex structures; in the building of these structures there is a layering of simple functions, each interfacing with others and using the capabilities of the layer previously supplied. Interestingly, the overall structure has proceeded to the point where the typical applications programmer knows little about the internals of the operating system he uses. To anyone trained in science or technology, this comes as no surprise; all of science and technology is the building of complex structures and theories from earlier simpler theories and structures. This is the Scientific Method.

Beyond the subroutine idea, the idea of the compiler quite naturally evolved. Since a larger number of subroutines were in existence, instead of the compilation of these subroutines manually into the program prior to running, it seemed natural to make the machine assemble all subroutines needed and at the same time create all necessary connective tissue to insure a coordinated, concerted data processing effort. Emphasis then shifted to the language—its use and its power. Attention also shifted to the development of assembly programs or compilers which would preprocess all programming information and create the running program. The programmer saw that, through the use of the subroutine-assembly program idea, he could write in decimal and let the machine carry out decimal-to-binary conversion. The programmer could write in mnemonic instruction codes and let the assembly program translate those mnemonic codes into numerical instruction codes. He saw that he could use symbolic addresses to make programming easier and have the machine later translate them to machine addresses by means of the assembly program. Many of these ideas were developed and explained by Hopper.[5]

Thus, the so-called "one pass" assembly program was born. This program performed the simple functions described above and used "regional coding" so that all machine codes would be assigned during the first pass through the translation process. There was no need for further recursive processing of the data. Incredibly simple by today's standards, these one-pass assemblers represented proud achievements by programmers in the early 1950's.

For a time in the development of system programming there was competition between interpretive methods and assembly/compiling methods. Assemblers and compilers took a source statement program and produced object code, properly location-keyed and ready to run. With a well-done assembly or compiler system, the object code could be efficient. The interpretive approach, on the other hand, used, in essence, "a different machine"—one built of programmed subroutines rather than machine language itself. Interpreters could be very rich in language power, but they tended to be profligate of machine time.

Today the place of the interpreter, except for special purpose interpreters, has been largely taken by the hardware/software design approaches of stored logic, read-only fast memories and microprogramming: "emulation" is the modern word used.

Early interpretive programming put heavy emphasis on applications in instruction and education in programming itself. Interpreters were used to structure simplified machines which could be learned easily and rapidly by the relatively uninitiated.[6] This design approach is seen in still viable form in many of the timesharing system languages of today. Among the early interpreters applied extensively was "Speed Code," an IBM project started in 1953 and featuring operations in floating point arithmetic.[7]

Throughout the 1950's and 1960's, the cost of machine time was always an important consideration. Thus, interpreters tended to remain unpopular. But near the end of the 1960 era, we began to see a rapid rise in the cost effectiveness of the computer. Today the interpretive approach enjoys a resurgence. Today's operating systems provide, through software, or a combination of software and hardware organization, a "new kind of machine," than previously seen and used by the applications programmer. The operating systems of today are closer to the interpretive approach than to the assembler/compiler approach.

Quite another facet of software came from the desire to have the machine assist in hardware and software diagnosis. One of the first of the systems stemming from the search for diagnostic power was the trace program. It was an interpretive program which printed the instruction itself and the results of the instruction if the instruction handled data. Since trace programs often required a long time in execution, selective trace programs were developed which printed only the data that the diagnosing programmer probably wanted to know.

One of the earliest of hardware diagnostics dealt with the problems of the memories of that day. Electrostatic tube memories suffered from having a particular storage location on the tube face affected by actions in surrounding locations. A spot on the tube face could spill over and affect neighboring spots—the "secondary emission phenomenon." Hence came the "read-around-ratio" testing programs. Such a program read around a given spot to determine the frequency of action at which failure would probably occur.

Other kinds of memory diagnostics took sums repeatedly over large segments of memory, assuring, hopefully, that the contents had not altered between successive summings. Similar to the read-around tests for tube memories are today's "delta noise" diagnostics for core memories.

Diagnostics led to an increasing realization that computers were strongly dependent on input/output devices. Early computer programs tended to be long on computation and short on input and output. Diagnostics put emphasis on input/output capability, and these devices too frequently proved to be the ones that needed diagnosing.

Pattern watching on computers with tube memories doubtless gave strong impetus to using a computer to control a cathode ray tube display tube. In about 1952, the famous "bouncing ball" output program was developed for display on output scopes (CRT) of the Whirlwind computer.[8] In this early work we undoubtedly see the first beginnings of on-line systems. For the first time there was immediate and visual reference to what the machine was doing. From this it was a simple step to let the programmer affect the operation of the program, based on what he sees, through the use of sense switches first (the IBM 704), later the keyboards of CRT terminals.

Machine aids to programming, or the design of machines easier to use, is not, in the strictest sense, part of the *res gesti* of the development of today's programming. Yet hardware and system software are so inseparably tied together today that we must consider the internal logic of the computer in a coherent discussion of software.

Among the earliest of additions to machine design was built-in floating point capability. We find, however, that such early, one-of-a-kind machines as the SEAC were equipped with an automatic trace mode.[9] The index register is a common design feature of computers today. It made its apparent first commercial appearance named as the "B box" on an early ElectroData Machine.[10] As old timers know, ElectroData machines later metamorphosed in Burroughs machines.

In the late 1950's, interrupt capability began to appear in computers. This was a most significant development. For the first time, the computer could be "impedance matched" with interconnecting noncomputer equipment. External devices could operate asynchronously from the main frame, and yet all essential communication was maintained.

Imaginative programmers soon developed many uses for the interrupt capability. What began as a narrow adjunct to early real-time systems was rapidly extended to general computer system operation; the interrupt was beginning to be used extensively to interface with common computer peripherals.

The programmer's propensity to interpret and emulate one computer on another through the years led to still another programming aid being built into the machine. Some machine designers reasoned that certain computers would operate more efficiently on certain types of problems if the instruction logic were chosen to correspond to that type of problem. Also they reasoned that emulation would be accomplished better if one could build up instructions from subinstructions—in other words, if there were access to subinstructions through programming. Thus, the idea of microprogramming was invented.[11] In its earliest form, it was not of great commercial significance nor did it receive immediate widespread use. Nevertheless, the important concept of storing in the memory the logic of the machine in such a way that the logic could be changed had a certain appeal and promise for the future.

The stored logic concept was brought into commercial use with the IBM 360 equipments which allowed emulation through microprogramming of second generation (pre-IBM 360) computers. Logical flexibility of the machine and ability to change its programming character, a kind of "subconscious" or "subliminal" level of the machine, has received the official stamp of approval. The idea was to be developed further in the 1960's through the use of privileged instructions through which only certain parts of the memory can be modified and, of course, the important step of the protection of the stored logic through read-only memories.[12]

Throughout the late 1950's and into the early 1960's, there was gradual increase in the recognition that the computer was neither a "scientific calculator" nor a "business data processor," but was, in fact, an information processing device. Scientific problems involving the solution of equations were a comparatively limited activity. Furthermore, scientific applications were usually imbedded in large design problems and engineering applications, problems which required general information handling capability. The recognition of this true character of the computer came about slowly indeed, and likewise the development of programming to reflect this true character of the machine. Programming systems for business data processing applications had been developed on a limited basis by Univac and by IBM in the 1950's, but the breakthrough in programming for business data processing applications did not come until the turn of the decade: in 1960 the COBOL language was specified and compilers were being developed to process the language.[13] An important new era had begun.[14]

PROGRAMMING LANGUAGES

In previous sections primitive programming languages, such as interpreters, assemblers and compilers were discussed. Later developments include programming languages which most directly affect the end users; i.e., those languages used for development of applications. This category includes computational and information retrieval ability on an *ad hoc* basis, as opposed to a predefined application program. Although *ad hoc* "programming" for one-time needs has certain similarities with the general category of program development, it is important to recognize that there are differences in terms of the user of the language, the facilities required, and the mode of usage (e.g., batch vs conversational, compiled vs interpretive, etc.). Basically, the "one-time" program should require less development investment than the continually productive program because of the short-term payoff involved. Thus, we trade off simpler programming language against efficiency of the object code produced. There are no prizes for *coding*; the results are what count.

Assemblers reached a language level approaching that of compilers with the facility for employing existing routines or macros. The macro assembler, in effect, permitted creation and usage of a higher-level language than that of symbolic assembly language. Nevertheless, assemblers produced object program code (instructions) which could modify itself and could be recognizable as source code by the originating programmer. Compilers, on the other hand, generated assembly language or object program code which was not easily recognizable to the source language programmer.

A principal goal of the "higher level" languages was to allow "machine independence" or, at least, to permit ease of transferability from one computer to another. The high point of this interest was reflected in the academic exercise with UNCOL (Universal Computer Oriented Language), as a means for machine-to-machine translation.[15]

There were many "languages" by the end of the 1950's, but only a few generally accepted by the user community for practical, operational use.[16] Needless to say, this acceptance is based on practical, *de facto* considerations, such as government or IBM endorsement and support. The acceptance of FORTRAN by the programming community was very significant because it helped reinforce subroutine library concepts, calling sequence standardization, etc. FORTRAN was designed primarily for scientific applications of numerical nature. The language was procedural with an emphasis toward algebraic notation and functions and the handling of matrices. FORTRAN did not accommodate parts of machine words or character manipulation.

ALGOL was a more definitive attempt at a computational language that was consistent with mathematically "clean" notation.[17] The ALGOL specifications generated a variety of subset language implementations, some of which still exist and are operating today. Although very useful for numerical computation, it lacked features required by business data processing, such as input/output facilities and report formatting.

COBOL was designed in order to satisfy the needs of the business data processing community, including an "easy-to-learn" language that was self-documenting, and the separation of data descriptions from the processing and operating environment.[18] PL/1[19] was aimed at a most comprehensive spectrum of programming applicability, scientific, business and system programming (including "real-time" interrupt processing). Because of its ambitious scope, it has taken awhile to gain its current operational acceptance. In the time-sharing environment, easy-to-use languages were developed, notably JOSS,[20] BASIC,[21] and APL.[22] JOSS was an interactive language system for small numerical problems. BASIC was designed as a simple but effective computational language for use by nonprofessional programmers in the conversational mode; its most notable facility was built-in text editing for creating and changing source code (which JOSS also had).

List processing languages, such as IPL and LISP, have been primarily used in artificial intelligence experimental and research work. The payoff is still to come in future machine architectures.

Each of these languages was designed to function with a compatible operating system but not with another language per se. (Some compiler implementations allowed for in-line assembly language code.) Thus, each language as a "system" had the following set of functional needs to account for, in addition to computational code generation and execution control needs:

1. data definition, file description—In order to properly handle various types of data and their logical storage organizations.
2. debugging facilities—Necessary for any application development effort at the *source language level.*
3. documentation—Required for proper maintenance of production programs and as a debugging tool.
4. output format control (reports)—This is an area of "programming" which is functionally different from computational programming.
5. Input/Output interface—Although there is supposed to be a maximum facility for "device independence," the realities of various input/output storage and terminal formats (e.g., typewriter or printer vs CRT) require proper handling.

In addition, a language system should offer different data entry, error reporting, recovery and debugging facilities for the conversational mode of operation vs the batch execution.

Every language system design varied in the manner and degree to which it offered these facilities and, in fact, every implementation had differences. What this proved was that programming language design was being accomplished at a time when the need was great but neither the operational environment nor the experience was stable and adequate enough to do a comprehensive job. Much mention has been made about the convergence of scientific and business computing requirements. However, it may have gone unnoticed that on-line (interactive) computing also pushed scientific computation to be concerned with things like character string processing, report formatting, etc. The established programming languages could not really "grow up" until the real world operational environment became more stable.

To cope with language deficiencies that appeared in order to program for new application areas, the idea of "extensibility" was introduced. This was really a takeoff on the macro concept and used in some compilers (e.g., ALGOL, PL/1). The extensible language approach, however, offered opportunity for language proliferation rather than standardization.

Although the above-mentioned languages were "higher level" for expressing computational requirements and easier than assembly language for program development, they did not adequately attack the complexities inherent with data access and storage, and another layer of simplicity was added in the area of secondary data storage or data file maintenance and handling. As the proliferation of machine files increased, both in numbers, usage and structure, a major gap appeared between the operating system, that was primarily concerned with physical data storage resources, and the programming languages that dealt with manipulation of data that the object program could get its hands on. It was up to the programmer to integrate all the protocol for getting the data in and out of files, along with appropriate file structure design and available access methodology. It was also necessary to be very *efficient*, because heavy input/output activity is the primary source of long and expensive machine runs. This kind of responsibility was taken over, in part, by the powerful *automatic* and *default* facilities of "file management" systems, which came into prominence about 1967.[23]

With the powerful automatic and default facilities, the development of *writing of* applications "programs" became much more simplified because most of the coding chores for doing things like validity checking, data file opening and closing, line folding report dating, column spacing, etc., were eliminated. It was not merely a case of providing facilities the programmer had to explicitly use; the use of structured forms and automatic functions meant that the applications developer had only to concentrate on the specifics of the application and was not forced to be a "systems" programmer. In effect, the automatic facilities of "file management" systems provides the kind of capability that was sought after in one of the historical buzz words, i.e., "implicit programming."

The descriptive name "file management" has been abused and confused because of the wide range of functions to which it has been applied. If any part of the data file accessing function were performed by a software package, it was called a "file management" or "data base" system. In reality, there can be complete languages which include all functional capabilities of, say, COBOL, along with file handling and sophisticated report generation facilities on the one hand, and on the other, file organizing and accessing supplements to application processing programs written in a standard programming "host" language. It is interesting to contemplate how comfortably the combinations will fit and evolve in the future.

OPERATING SYSTEMS

Concurrent with the goal of making programming easier for the professional programmer, was the pressure to make computer operations more efficient and reliable. These two major objectives gave rise to the modern day operating system.

Operating systems, monitors, control programs, or executives, etc., became more necessary as computer hardware became more complex; the requirements for sophisticated operating system functions depended largely upon evolving machine architectural facilities. The operating system, as a software facade for the hardware, had to exploit the strengths of machine design as well as make up for deficiencies.

The concept of operating systems has taken computer usage full cycle from the open shop, job-at-a-time approach to the batched, closed shop environment, and back again to a flexible combination of on-line, time-shared execution coupled with batched production jobs. In similar manner, we have seen a pattern of independent computers, followed by complete centralization, and now distributed computers in a network approach. Needless to say, the extremes of open or closed shop operations did not represent a true picture of the needs of the real world. Although there were a number of special purpose systems, particularly command and control systems that utilized advanced operation system ideas ahead of their time, it has taken awhile for these operational concepts to evolve in the commercial world, and they are still evolving along with hardware and software developments.

The early days of job execution in the 1950's found things done on a single-program-at-a-time basis. A programmer personally, or through instructions to the computer operator, supervised program execution, directing all input/output routines with his own code, coding specific peripherals, and tying up the complete computer configuration.

In the interests of efficiency, the closed shop kept machine operations in "specialized" hands and thereby helped to increase throughput. However, job setup time, during which the computer was in an idle condition, still was a problem. This was true because of the lack of permanent auxiliary storage; all jobs had to be read singly and directly into the machine. Even with the use of magnetic tapes, there were delays in setup time and tape changes. The evolving operating systems were aimed at automating the sequence of job execution.

The relative slowness of peripheral input/ouput equipment; i.e., card readers, punches and printers, compared to magnetic tape, pointed to the next area of throughput efficiency. Thus, in the mid-1950's, early batch monitor systems (see Figure 2) were used as simple loaders to control the sequence of jobs and program processors read in from magnetic tapes.[24] These monitors were themselves self-loading from tape. Primitive "multiprogramming," the overlapping of program execution with input/output, was accomplished by off-line use of card-to-tape and tape-to-printer equipment.

The availability of magnetic tape made possible new benfits to ease the chores of the programmer. By standardizing input/output to system tapes, the use of common library input/output routines was encouraged. This sharing was particularly promoted with the advent of FORTRAN where such facility was conveniently made available. The use of magnetic tapes also escalated the subroutine library concept in a dynamic way: user programs could capitalize on the availability of library routines loaded from a library tape when the user program was loaded. These routines were useful only if flexible enough to be used anywhere in the object program. Finally, magnetic tapes fostered the overlay concept by allowing portions of a job, too large for the available core memory, to be brought into core when called by the executing program. This concept expanded the role of the loading function of these primitive monitors.

The impact on programming languages became considerable. The loading function and calling sequences, as well as availability of certain functions at execution time, required language development compatible with this operating system environment.

Hardware constraints on computer operations; i.e., core size, lack of sophisticated independent input/output, direct access secondary storage, etc., all tended to keep computer operations limited to one job at a time in the main frame. As long as this was true, there was little need for sophisticated accounting, security protection for files and hardware protection and control of machine resources. However, toward the end of the 1950's, the advent of independent input/output channels pointed to the use of multiprogramming concepts and concurrent usage of machine resources to increase throughput efficiency.

In order to take advantage of and service the new I/O hardware facilities, it became necessary to install a resident software package called an I/O or interrupt supervisor. On the 7090, there was IOCS and it provided I/O-CPU overlap and core buffering to keep the CPU busy. Service does not come free; one of the penalties of the I/O supervisor was the preemption of scarce core memory. Old programs had to be converted to use the new environment. All of this contributed to a somewhat negative attitude on the part of many of the early assembly language programmers toward monitor systems.

Whether the programmers liked it or not, it was necessary to preserve the integrity of resident routines. Thus, software methods for protection were adopted. Since it was possible to violate the resident routines and tables through erroneous I/O requests, it was appropriate to check the call requests for legitimacy prior to execution. I/O error conditions had to be properly handled so that object program execution could be gracefully terminated. Many a running system died because a user program branched into or stored data in the supervisor area and ran amok.

With the repertoire of execution time services provided by resident monitors, a new language appeared. This was for control of job execution (704 Monitor), handling such matters as peripheral device assignments, job accounting, job abort handling, and loading information. While job control languages have been the source of sophisticated processing procedures, they have also been the source of much waste of time and machine resources because of sensitivity to erroneous usage.

In the early 1960's, computer architecture took a turn toward "real time." This involved a real-time clock, the heavy use of interrupts, and the increased need to make the resident operating system inviolate. The addition of hardware protection, boundary registers, provided safeguards for sensitive areas of memory.

Although drums were available for fast secondary storage, their limited size and high cost inhibited wide use. Disk files provided high volume storage which was faster than tapes and less expensive than drums. Disk files made practical the use of resident monitor systems which could be overlaid from disk and also provided faster loading from the subroutine library.

Most significantly, multiprogramming techniques became highly practical in several ways, based on fast access drums and high capacity disks. The multiprogramming approaches involved staging peripheral input/output via disk, or spooling. This permitted faster program execution because all input and output occurred via disks rather than the slower card readers or printers. Another variation of this approach was the direct coupled system or "moonlight" system (e.g., 7094/1440) where the main computer performs the computation and the smaller computer controls the peripheral I/O, using a disk. Last, but not least, multiprogramming was applied to on-line interactive computer usage.

The pioneering general purpose time-sharing systems such as MIT's CTSS[25] and SDC's Q-32 TSS[26] highlighted the next round of facilities and responsibilities for modern operating systems that evolved during the latter part of the 1960's. Hardware protection was augmented to include Master-Slave modes of operation which enabled only the resident operating system or other privileged programs to execute I/O and certain control instructions. User or problem programs are trapped if any attempt is made to execute such privileged instructions. The operating system also had to provide access protection for centralized files.

With the shared concurrent use of system resources, such as core, disks, and I/O channels, the old wall clock method of computer usage accounting became obsolete.[27] Now it is important for the operating systems to account more specifically for the resources used, e.g., core memory, I/O, CPU time, disk storage and communication lines. This accounting capability has become very critical for proper operational management and configuration tuning.

The capability of relocating loader functions of operating systems increased with the loading of object programs via permanent secondary storage. The ATLAS computer in England played a great pioneering role in using secondary storage concepts.[28] Such hardware developments as memory maps for automatic relocation or virtual memory machines to eliminate the need for program overlay structures are currently impacting programming design. The concurrent usage of a program by several users in the time-sharing environment has given greater emphasis to potential savings from reentrant coding or pure procedures where all modifiable data or variable data is separated from the reentrant code. Job scheduling has become more intricate in terms of concocting priority schemes and scheduling algorithms for maximizing throughput or optimizing turn-around time for particular job classes (especially conversational processing in the time-sharing environment).

With the on-line multi-user system, the need for reliable operation became critical. As a result, recovery and restart facilities were added to operating systems. There is a great difference between rerunning a user job by the batch operator and having a large number of terminal users banging away at dead keyboards.

Communications handling by the operating system became more standard for batch work and for conversational terminals.[29] The marriage of data communications with data processing, among other benefits, permitted the effective implementation of centralized data bases, so critical to the needs of the modern business world. This in turn, has emphasized the need for separation of processing routines from data structures in programming design and development.

There are other practical benefits from the communications interface in terms of the network concept. The ARPA network, for example, is an indication of the linking of independent systems, many having "specialties." A user at a terminal can have access to pertinent data and processing capabilities other than his own.

The current trend of computer operations is leading away from restrictive batch production only toward a more flexible operational environment. Hands-on interactive facilities are of growing importance. The modern operating system also offers the in-between approach of conversational remote job entry which can exploit the effectiveness of both batch and interactive modes of operation. The evolution of operating systems has been one of cumulative growth and expansion of facilities along with control responsibilities. This growth can be traced in Figure 2. Future growth appears to be limited to consolidation and refinement of the current scope of functions with major emphasis upon hardware/software architectural synergism.

ON-LINE SYSTEMS PROGRAMMING

From the earliest days of development of the computer, consideration was given to electronic systems in which the computer was an imbedded piece, usually the control element. The early systems were those for the military, such as fire control and air defense systems.[30,31] However, it was not until the early 1960's that on-line systems* became sufficiently active to have a definable

* We use the term "on-line systems" to refer to all systems in which the computer is attached to instrumentation from which it receives (and gives) signals asynchronous to its operation. This terminology, we find, is superior to "real time." "Time-sharing" systems refer to those on-line systems in which many users are "simultaneously" using the machine.

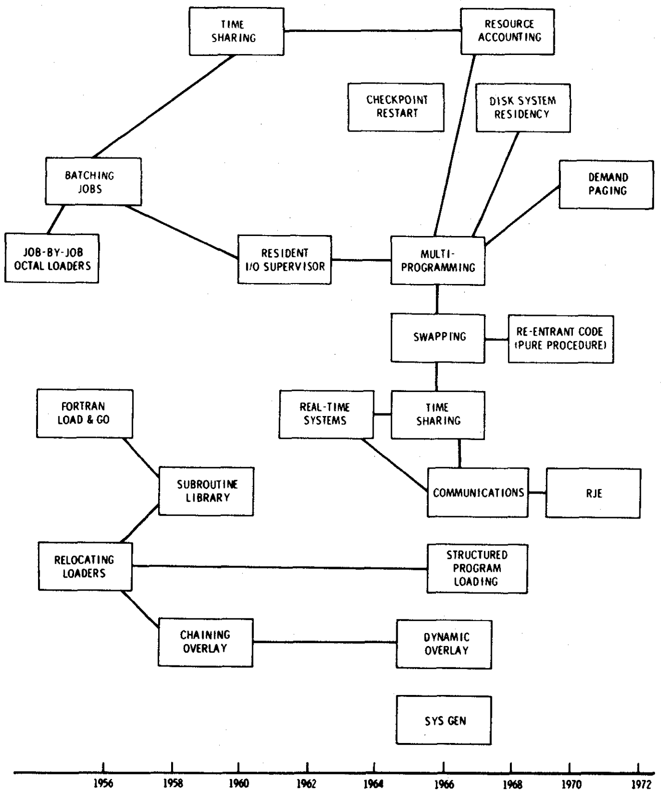

Figure 2—Evolution of operating system functions

programming structure.[32] By the late 1950's, the interrupt feature was "invented" and appeared on large scale computers such as IBM's Stretch and Univac's LARC, introduced in the early 1960's. Programmers were increasing their understanding of the relationship to interconnecting instrumentation and were becoming better able to cope with it. In the early 1950's, the information was given to the card reader and the program "waited" until the entire card was punched. By the late 1950's, with the introduction of computers such as the IBM 709, the programmer was doing useful computation between punching of successive card rows. The interrupt feature was important in cases such as this; it allowed the program to continue the processing, ignoring the operating of the external devices (and not having repeatedly to test its status), and wait confidently for the interrupt to occur to cause control to be transferred to the interrupt handling routine.

By the late 1950's, the subject of "multiprogramming" or "parallel programming" was being investigated.[33] This approach was aimed at better machine efficiencies through the overlap of various machine functions hitherto done purely sequentially. Computation could well proceed while memory cycles were "stolen" to complete an input-output process. Also, the approach of tying computers together ("multicomputers")[34,35] was receiving increasing attention. These techniques, aimed at more efficient machine operation in increasingly complex application environments, required new programming solutions. Control programs were needed to keep all the units operating efficiently in concert.

An important subset of on-line systems is the computer/communications system. The development of these systems has been traced, and the system defined elsewhere.[36] The development of particular functions such as network control, message handling and line control has resulted in an important programming specialty, and one of growing importance.

The beating heart of the on-line system is the "Master Control Program," sometimes referred to as the "executive." This Master Control Program was a unique, definable entity for it had no counterpart in general purpose computing. The MCP performed the functions of scheduling, controlling, synchronizing and monitoring the entire process. It took its place alongside of and made use of utility programs (data moving, memory controlling), console programs (the data formatting, controlling operator logic), and application programs.

By 1962, the subject had developed to a point where a definitive paper[36] could be written on the subject. Subjects which were now clearly understood, and hence amenable for further development and more extensive application included: man/machine interaction via consoles, master control program design, multicomputer and "graceful degradation" aspects, and comprehensive interrupt handling.

General purpose operating systems and their application programs undoubtedly benefited from the on-line systems development that started with the military systems. Time-sharing, interactive systems, communication systems, the modern operating systems, and the world of minicomputer applications can be said to have these on-line and multiprogramming concepts as direct antecedents. If one looks at the character of the on-line systems (mostly military) developed prior to 1962, one sees that these modern day systems perform many of the functions first performed and provided in these earlier systems.

EVOLUTION OF SOFTWARE PRODUCTS*

The notion of software as a product came late. This was probably so because of the program-sharing spirit of early computer pioneers. Most of these people were

* Sometimes called "program products."

of the academic or scientific viewpoint. They believed in immediate publication and full disclosure of all developments. Further, there was no thought that software could become a commercially important product. The tradition of freely shared software was fostered by organizations such as SHARE, USE, and GUIDE.

It is hard to name the first significant instances of software sold for money. Codes for linear programming were developed by joint ventures in the late 1950's. They may have been the first proprietary software sold in the United States. Linear programming is important to the work of the petroleum industry. Companies in this industry developed linear programming packages either singly or in coalition. Some commercial software and service companies developed their own proprietary methods and sold the services of these, though they rarely sold the software itself, since the demand was insufficient.

The mid-1960's was a period of controversy about software products. Some spokesmen proclaimed the bright future of software products as early as 1962, while other pundits as recently as 1968 said there could be no such thing. Much of the confusion about software products can be attributed to a lack of understanding about what kinds of software could be "productized." General-purpose "tools," which can be utilized as-is by users, make the ideal software product. Application packages, on the other hand, particularly those which are data base dependent, will usually require "tailoring" before they are of practical value to individual users. This is true because of the lack of standards in the specific application areas and the dynamics of real world needs.

The economic leverage of the software product is too great to be ignored forever. In view of ever-increasing machine capability and dramatic increase in computer cost effectiveness, the path to ever-increasing software complexity was wide open and lined with roses. Sooner or later the world had to face the high cost of completely home-grown, custom software. The distressing shortage of highly competent software system designers and system programmers did nothing to improve the picture.

The resulting flourish of software entrepreneurs blossomed remarkably rapidly. Software companies sprang up everywhere. They were, in many instances, encouraged by the view that the software product has miniscule manufacturing costs, except at first item delivery. The price bidding could be against the cost of a purportedly equivalent in-house product, and the software supplier might indeed come out looking pretty good.

Some of the brave hopefuls among the many small and large software companies did not survive to see their planned products delivered. Some of them had badly underestimated the development costs of the first delivery item. Others presumed that no manufacturing costs meant no marketing costs. But the most consequential and glaring mistake was the significant underestimation of maintenance and improvement costs and general post-sale customer support.

By the mid-1960's, few if any data processing groups would consider in-house development of a compiler. Also, by that time the idea of developing any significant piece of systems software or programming tool on a one-of-a-kind basis was becoming suspect.

Software products became real for all the world to see when IBM, largely in response to anti-trust pressures, announced unbundling. Remaining doubts about the reality of software as a product quickly disappeared. The 1972 Datamation Industry Directory lists 41 pages of them in 119 categories.

About the time of IBM unbundling, there were many predictions about software products, some rather ill advised. It became a mistaken belief, particularly among starry-eyed startup investors, that any piece of software which purportedly worked was "a software product." There were some bad financial losses and disappointments and a pervading mistrust of software products for a variety of psychological reasons. Fortunately, both time and need are overcoming the mistakes of the past.

While the patent situation is still unclear with regard to software, lack of patent protection will not be a deterrent to future developments of software products. The trade secret body of law, and binding contracts with customers and with employees, afford ample protection for the software products supplier.

It is interesting to note the differences between software products and hardware products. Design and development is expensive in both areas of business. Maintenance is highly decentralized with hardware, but much more centralized with software, since the change need only be made once, then copied to all users. Software has essentially no manufacturing costs. Marketing costs are high for both hardware and software. As a percentage of revenue, marketing costs are currently higher for software than for hardware.

FUTURE TRENDS

The central part of this paper traced the developments of programming during the modern short twenty year history of modern computing. On the theory that we learn from the past in order to better equip for the future, it seems appropriate in the light of the generated historical perspective to comment on the future of programming. These comments take the form of opinions on the current state of programming matters and predictions for the future.

The day of programming languages being developed by the hardware manufacturer passed in the late 1950's or early 1960's. COBOL, the last language to be developed which has in any sense gained universal acceptance (in the business world), is now more than ten years old. FORTRAN, still going strong, is even older. The last attempt at a more "universal" language, PL/1, has been something less than a spectacular success. There is nothing on the horizon to suggest the development of a single comprehensive language for all scientific and business applications.

The programming profession has apparently come to realize that there is no such thing as a "universal language." It has probably also come to realize that there is a fundamental conflict between the power of a language by a wide class of applications on one hand and the universality of that language on the other. Languages in the framework of a burgeoning set of application "areas" cannot be all things to all users.

It is appropriate to distinguish between a computer-based application which processes a specific set of input parameters and produces a specific set of outputs and an application "area" such as numerical control, photocomposition, computer-aided instruction, etc., where a whole class of problems including unique terminology and functions, must be dealt with in a flexible manner. The latter requires a special "problem-oriented" *language*, while the former needs a specific application *program*.

The software era that we see the industry entering is a phase of maturation that reflects both understanding of real world needs and technological advances. Software will be separable into three major areas:

- Tools for the development of production applications; e.g., language compilers, file menagement systems, utility packages, etc.;
- Application programs for the direct use of *end users* (or industrial processes). This will also include more general purpose language systems for simple *ad hoc* inquiry and computation; or,
- Operation control programs—operating systems that control execution of user programs in an increasingly complex machine environment.

In the area of scientific applications, the major advances appear to be the incorporation of those computational and manipulative functions (such as array manipulation) that the computer makes so convenient into language notation, as has been done with APL. However, the scientific community has been reasonably satisfied in terms of language needs for defining their problems. "Problem-oriented" languages will continue to be developed as well-defined application areas with unique requirements providing some basis of standardization.

The pressing needs of the business world, however, which are more pervasive and have been more neglected than the needs of the scientific community, will further help to establish the concept of data structures that can be independent of program structures. Data and file descriptions will permit data access and manipulation to be accomplished by application programs dynamically at execution time. There will also be an expansion of facilities for the proper management and control of data within an operational system.

It would appear that in the foreseeable future, language development for business data processing will consist of two mainstreams . . . COBOL-oriented preprocessors (including shorthand and optimizer preprocessors) and the file management system approach. The COBOL generator approach seems to be consistent with evolving techniques throughout the history of programming: layers of capability are successively added to the machine in the forms of software systems. The operating system relies heavily upon assembly language. The COBOL layer is developed upon this combination of hardware and software. Shorthand languages are developed to facilitate the preparation of COBOL programs. It is clear that with the need for stability and continuity in business applications, the preprocessor approach seems destined to become increasingly important in the future.

In earlier sections of this paper, the reader has seen how the file management approach evolved from the early report generator systems. It seems clear that these generalized data management systems represent the only possible claim to a language (or a type of language) to compete with COBOL. In the realm of *ad hoc* (one-time) retrieval, these systems have a clear-cut superiority over COBOL, and all application areas have a need for such a capability. These packages are becoming increasingly powerful and increasingly used. The trend will undoubtedly continue. There will be a profusion of enhancements encompassing complete data base needs and resulting in systems with faster programming execution and manifold options with which to tailor systems to the particular application needs of the user. COBOL achieved some degree of universal acceptance because it was developed during the days before software was considered a business in its own right and because it had the power of the Department of Defense backing it; the file (or data base) management systems being developed today are proprietary items being developed by private industry and, although no single version among them is likely to achieve industry-wide endorsement, as a class, data management systems are likely the wave of the future.

Programming languages for application development will consist of not merely the compiler or code generator, but a complete language-compatible set of tools for debugging, maintenance, documentation, etc. Thus, we must think of a language "system" which starts with application design needs through production operational requirements.

The concept of applying "layers" of language will become most significant in the area of applications. By definition, every application program creates a new king of language; i.e., its data input format. This is true for highly parameterized inputs and for more flexible, *ad hoc* query and computational languages for specific applications. The preprocessing approach is most effective when a highly interpretive mode is needed for an interactive phase of operation or for very user-oriented, automatic code generation, *and* the "under layer" language is an efficient standard. Compilers are frequently preprocessors for assembly language; application preprocessors for compilers and interpreters, particularly for business use, are coming into wider usage.

The need for simplifying the programming effort will promote the use of automatic and default programming tools. This has proved particularly successful in business applications where common needs are stabilizing and coding ingenuity does not produce much practical payoff. It is not sufficient to provide tools that require sophisticated expertise to avoid mistakes and misuse. It must be there if necessary, but the common need can be handled in a more simple, automatic fashion. The file management and report generator packages have notably demonstrated the effectiveness of this approach.

The development of comprehensive operating systems is relatively recent, considering that the first such "running operating systems" really came into being with the third generation computers in the mid-1960's. We have seen only the beginning of what will undoubtedly be an exceedingly important development in the years to come. The extraordinarily high speed of the modern computer, the relatively low cost of high speed memory, and in general, the dramatic increase in cost effectiveness, means that complex layers of "machine" capability (layers of software, actually) will be developed, making the job easier for the programmer at the application or the compiler level. The operating system developer in reality is building a new machine on the basis of stored logic principles starting from the standard machine instructions. The trend is clearly evident over a decade. There is no question that it will continue.

The role of operating systems has become significant enough to warrant the concern over "operating system independence." It is obvious from a practical standpoint that computer operations must be sustained for the real world production environment. Not only must there be no drastic interface changes, but there must be graceful, upward compatibility with new facilities in the areas of communications, data structures, storage devices, etc.

It is apparent that the general purpose operating system will allow the user installations a wide range of operational modes ranging from batch, remote batch, to highly interactive as well as transaction-oriented, online usage. The distributed network of computer processing will also expand remote access and the concept of computer-based services.

Because of the increased operational usage and the varied response requirements, the proper management of configuration resources will become most significant. Thus, instrumentation for evaluating throughput performance, coupled with controls for dynamically allocating priorities and resources, will assist in the understanding and the effective management control of day-to-day operations.

The complexity of operating system facilities which has caused the command language or job control language to proliferate, will cause more user-oriented, automatic generation of such operating requirements to be "buried" with the applications program.

The last ten years have also seen a marked increase in the appreciation of cost of complex software systems. This is leading, in turn, to a most important trend which does not seem to be clearly recognized. In the past, whenever the programmer received a new machine, his immediate and first thought was to reprogram his application so that it would operate on the new machine. Now that many applications are more stabilized and the high cost of programming systems more clearly recognized, the hardware will be brought with increasing frequency to the software rather than vice versa. This means that the emulation by one machine (presumably the more technologically advanced one) of another machine will become increasingly important. This is mainly accomplished through microprogramming and stored logic techniques. We have seen this approach taken in the important commercial arena with the third generation computers. We have seen the idea advanced through hardware with the use of read-only memories and other microprogramming techniques. As long as cost effectiveness of hardware increases at the rate it has in recent years and cost effectiveness of software approaches improves much less rapidly, there will be increasing pressures to bring the hardware to the software.

Hardware approaches to make the programmer's life easier will proliferate. In the mid-1960's, we saw the first machines of commercial importance expressly designed for time-sharing. These systems included hardware and software implemented capabilities which allowed the programmer to prepare his programs as if he were the only user and as if he had unlimited memory. The latter involved so-called "paging" and virtual memory hardware facilities. The advent of these systems underscored the importance and acceptance of resource-sharing techniques in the industry. It is likely that by 1980, even with bigger, cheaper core memories, nearly all new machines will have these capabilities and further relieve the applications programmer of the concern for core storage limitations.

Application packages as proprietary software will become an important commercial area. Until now, the customer of (or prospect for) an application package has been unwilling to accept the system except in the exact form he conceives it, all the way from computing algorithms to the form of output reports. On the other hand, the supplier community has not developed techniques which allow the creation of flexible, automatically tailored application software. This gap will narrow; the customer community will become more tolerant in view of the economics of the situation, and the suppliers will design and build more flexible software.

Utility packages (sorts, compilers, data management systems, etc.) will continue to grow as important proprietary software.

CONCLUDING OBSERVATIONS

We cannot let the opportunity pass to wax philosophic about programming.

Senior observers of the last twenty-five years of programming development all agree on one point: it is a continuing shock and surprise to realize anew the span of time between the development of an idea and its widespread, practical use. Old-timers recall how slow the programming professional was in 1958 to embrace FORTRAN. In 1964, why weren't almost all business applications being programmed in COBOL? The lack of fully checked-out, fully compatible systems is not the answer, although such system problems contributed in a minor way. The answer is that there seems to be an inherent conservatism within the professional programmer working in business and industry. Is he subconsciously increasing his job security? Is the spectrum of progressiveness and inventiveness extraordinarily wide within the programming profession? Is the lack of proper education at fault? Finally, should we blame the dominant computer manufacturers for *de facto* control of progress?

But let there be no doubt about the fact that change has occurred.* The developments recorded on these pages are mute but uncompromising evidence of that. Nevertheless, it is disappointing that, in the estimate of most, programming cost effectiveness improves slowly, probably 5 percent per year compared to a whopping estimated 25 percent for hardware. Perhaps this is due to immutable, intrinsic characteristics. But it may be a result of the fact that the data processing industry has placed true programming at too low a professional level. The lack of attempts within the profession to structure the field and organize and discipline programming activity are also contributing factors of a more minor nature.

REFERENCES

1 J A POSTLEY
Computers and people
McGraw-Hill New York 1960

2 W S MELAHN
Description of a cooperative venture in the production of an automatic coding system
Journal of the ACM Vol 3 No 4 November 1951

3 S ROSEN
Programming systems and languages
McGraw-Hill 1967

4 M V WILKES D J WHEELER S GILL
The preparation of programs for an electronic digital computer
Addison-Wesley Press Reading Mass 1951

5 G M HOPPER
The education of a computer
Proceedings of the Conference of the ACM Pittsburgh 1952

6 JEAN E SAMMET
Programming languages: history and fundamentals
Prentice-Hall Inc Englewood Cliffs NJ 1969 pp 12–13

7 J W BACKUS
The IBM 701 speedcoding system
Journal of the ACM Vol 1 January 1954 p 4

8 R R EVERETT
The Whirlwind I computer
Electronic Engineering 71 August 1952 pp 681–686

9 S GREENWALD R C HOUETER
S N ALEXANDER
SEAC
Proceedings of the IRE Vol 41 (October 1953) pp 1300-1313

10 *Commercially-available general-purpose electronic digital computers of moderate price*
Proceedings of the Symposium of the Navy Mathematical Computing Advisory Panel and the Office of Naval Research Washington DC May 14 1952

11 M V WILKES
The best way to design an automatic calculating machine
Manchester U Computer Inaugural Conference 1951

* The twenty-fifth anniversary edition of the Communications of the Association of Computing Machinery[38] was published too late for inclusion as a major source of references for this paper. However, it has many valuable articles which relate to the topic of software history and trends.

12 R F ROSIN
Contemporary concepts of microprogramming and emulation
Computing Surveys ACM Vol 1 No 4 Dec 1969

13 *COBOL: initial specifications for a common business oriented language*
Department of Defense US Government Printing Office Washington DC April 1960

14 R BEMER
A view of the history of COBOL
Honeywell Computer Journal Vol 5 No 3 pp 130-135 1971

15 J STRONG et al
The problem of programming communication with changing machines: a proposed solution
Comm ACM Vol 1 No 8 1958

16 SAMMET Op Cit p 5

17 A J PERLIS K SAMUELSON (for the committee)
Preliminary report—international algebraic language
Comm ACM Vol 1 No 12 Dec 1958

18 E L WILEY et al
A critical discussion of COBOL
Annual Review in Automatic Programming Vol 2
Pergamon Press New York 1961 pp 293-304

19 *IBM system/360 operating system: PL/1 language specifications*
IBM Corp C 28-6571-0 Data Processing Div White Plains NY 1965

20 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
Proc FJCC 1964

21 J G KEMENY T E KARTZ
BASIC
Dartmouth College Computation Center June 1961

22 K E IVERSON
A programming language
John Wiley & Sons, New York 1962

23 J A POSTLEY
The MARK IV system
Datamation January 1968

24 C L BAKER
The PACT coding system for the IBM type 701
Journal of the ACM Vol 3 No 4 October 1956 pp 272-78

25 F J CORBATO et al
The compatible time-sharing system, a programmer's guide
MIT Press Cambridge Mass 1963

26 J I SCHWARTZ E COFFMAN C WEISSMAN
A general-purpose time-sharing system
Proceedings of the Spring Joint Computer Conference 1964

27 A M ROSENBERG
Computer usage accounting for generalized time-sharing systems
Communications of the ACM Vol 7 No 5 1967

28 T KILBURN B G EDWARDS M J LANIGAN F H SUMNER
One-level storage system
IRE Transactions April 1962

29 A M ROSENBERG
Group communications in on-line systems
Proceedings of On-Line Computing Systems Symposium UCLA/Informatics 1965

30 E H GOODMAN
Sage
Computing News Vol 6b Nos 15–17 Aug through Sept 1958

31 W F BAUER W L FRANK
Doddac—An integrated system for data processing, interrogation, and display
Proceedings of the EJCC December 1961

32 W F BAUER
On-line systems—Their characteristics and motivation
On-Line Computing Systems (Proceedings of the Symposium) America Data Processing Inc Detroit 1965 pp 14-24

33 S GILL
Parallel programming
Journal of the British Computer Society 1957

34 R PERKINS W C McGEE
Programmed control of multi-computer systems
Proceedings of the IFIP Congress—1962 Munich Aug-Sept 1962

35 W F BAUER
Why Multi-Computers?
Datamation August 1962

36 W F BAUER
Computer communication systems: patterns and prospects
(Proceedings of the Symposium on Computers and Communications) Toward a Computer Utility Prentice-Hall Englewood Cliffs New Jersey 1968

37 W L GORDON G L STOCK
Programming on-line systems
Datamation September 1962

38 *Communications of the Association of Computing Machinery*
Vol 15 No 7 July 1972