Do-it-yourself software experience

By B. L. Neff*

This paper is based on a talk given to The British Computer Society in London on 4 May 1965. It records experience in writing a series of compilers to accelerate the programming work of a large commercial application.

My subject is software, and by that term I do not mean programming in general, but only that phase of programming of a "service" nature: compilers, executive routines, sort programs, debugging devices and the like. To further specialize my subject, I am with a Company whose computer work is almost entirely in the "commercial EDP" area, so my remarks will have only incidental application to scientific programming problems. Furthermore, I speak to you with the viewpoint of a *user* of computing equipment, in a Company that looks upon electronic equipment as a necessary, but not evil, device for saving money and rendering better service to our customers.

I think it important to describe what might be called the "environment" of our installation. I have attended computer meetings where apparent disagreements between speakers could have been avoided if each party had realized that the computer environment of the other was such that quite different ways of solving problems were separately correct, each in its own environment.

The Metropolitan Life Insurance Company is the largest Insurance Company in the world. Our assets are exceeded only by those of the U.S. Bell Telephone System, and we insure over 45,000,000 people in the United States and Canada. We have a lot of records to keep up to date, hence the situation could not be more ideally suited to computer usage. We are in a business whose only products are in paper form: policies and cheques. So I would list environment condition (1) as: a tremendously large potential computer application.

We started out in the electronic computer business in 1954, if one counts the installation of actual hardware as the start; obviously many prior years of study were necessary. Once the initial installation had proven itself, the Company's attitude of tentative experiment gave way to a large flashing green light that said "go-go-go." It is now realized that potential savings can be converted into real savings, and we are going ahead as rapidly as possible. But our business is in the nature of a public trust, and we cannot be rash. Hence environment condition (2) is: Plunge ahead cautiously: *or* Full speed ahead, but be careful.

Another environment condition is the fact that we are now in our second go-around with a large-scale system (first Univac I and II, then Honeywell H-800, H-1800) and hope we have acquired some sophistication in the process. Still another condition important to mention is that we do not have too much trouble hiring and retaining new people who are suited for training primarily in the insurance-applications end of computer work. The job market for machine-oriented programmers on the other hand is a bit unstable at times, depending somewhat on the condition of government contracts, new machines being announced, etc.

The job of programming insurance applications, with emphasis on knowledge of company practices and procedures, will probably become a stepping stone toward assignment in higher management positions. This makes it possible for us to hire good people who may be attracted by this prospect. Obviously the future of a machine-oriented programmer, however good he is, is more restricted in scope. Hence we have a need for a programming language that bypasses machine complexities and is problem-oriented.

Hardware now in use

The actual hardware now in use is pertinent to my discussion. In 1960, Mr. John Finelli of my Company addressed this group (Finelli, 1960) and gave our earlier history of computer utilization, since 1954. In 1960 we had reached the point of using three Univac II machines on our New York premises, and renting time on two others at outside locations. A fairly exhaustive study was made in 1960 concerning our next "generation" of machines, and the end result of a long series of meetings was the choice of the Honeywell H-800. Two were installed the following year, and at the present time the New York Home Office contains two H-800's and two H-1800's. Another H-1800 will be installed by July 1965. Each of these New York machines has 30 tape drives and 32,000 words of memory. Two of them have five Univac tape drives. Our San Francisco and Ottawa Offices each have an H-800, with 10 tape drives and 16,000 words of memory.

Although it has little bearing at present on software questions, I should mention, for the sake of completeness, that we are well along the way to installing a country-wide data-communications system, in the United States only, to link 800 District (sales and service) Offices with the New York computers. Ten districts should be operating on this basis early in 1966. It is not a true real-time inquiry system in the sense of instant response, but rather a method of collecting and disseminating data

* *Assistant Vice-President, Metropolitan Life Insurance Company, 1, Madison Avenue, New York, N.Y., U.S.A.*

daily, with daily data-processing of inquiries, etc., to enable us to reply, when necessary, by the next day. An H-1800 in New York will scan the country automatically, with simultaneous sending and receiving on multiple lines.

This concludes my discussion of the environment conditions in the Metropolitan's computer installation.

Early programming experience

To return to the software story: those of us who did the original programming on Univac I in 1953 and 1954 started out being thoroughly indoctrinated with the virtues of machine coding. In 1953 there wasn't much of anything else anyhow. In 1954 we were very fortunate in getting a "compiler," produced at New York University by Roy Goldfinger. It amounted to what we would nowadays call an *assembler*, in that it allowed subroutines to be stored on a library tape, to be called in by a programmer at assembly time. We recoded it in our shop to add some features we thought desirable, later recoded it again for Univac II, and it is still one of the basic Metropolitan assembly systems for Univac II programs. It is a sobering reflection on the fantastic progress of computer technology in the last 10 years to remember how avant-garde this assembly system appeared to us in 1954, and what a great help it was in our programming work; then to come back to the present day, when an assembly system is taken for granted, like a self-starter on an automobile.

A few years after our 1954 start, the first ideas about using problem-oriented languages for coding were being discussed, advocated, sneered at, and praised. We programmers thought these efforts quite impractical, but our higher management saw them as rays of hope on the horizon. Systems and claims for systems that used actual English language words came into being: *B-ϕ Flowmatic*, *COBOL*, *FACT*. We tried out the B-ϕ system on our own machines. The early efforts at implementing these languages on the small-memory machines then available produced results that were such that we had no trouble recognizing their impracticability. But these systems had a point which was inescapable: the country simply did not contain enough clever machine-oriented programmers, or people who could be so trained, to cope with the explosion of computer programming that was occurring. So a few of us stopped scoffing, and considered that we didn't really object to the general *language* approach of these new systems—it was the *implementation* we didn't like, i.e. the volume of coding generated, and its efficiency in terms of running time of the generated program.

First compiler

In 1959 therefore we decided to make up our own English Language Compiler for the Univac II, designing the language ourselves by considering real existing programs, and requiring it to handle everything we could think of. Before the language was frozen, it was applied to actual production runs, large and complex, to establish its practicability. We were conscious at the time that some new, as yet unknown, machine would replace Univac II within a few years, and tried to keep the language reasonably machine-independent.

This first effort in compilers in our shop was working by the end of 1960, and the coding produced was not bad. A basic decision was made during the implementation phase, which we have held to in our later compiler work, and which I consider important: it was decided that the efficiency of the coding produced was far more important than the efficiency of the compiling process. That is, we did not care whether compiling time was fast or slow, but the object program produced had to run fast, and had to use memory space economically. The coding produced had to compare favourably with direct machine-language coding by a human being, although we realized it could never be quite that good. At that time, and you can still see it today, claims of competing compilers seemed to emphasize compiling time exclusively: "Our compiler produced this program in only three minutes, while the Brand-X compiler took five minutes." The Metropolitan Univac Compiler took about an hour, sometimes longer, to produce a full 2,000 word program. This is one of the reasons why I stressed earlier in my talk the importance of considering the environment in which software is used before evaluating it. Our data processing activity features many recurring jobs, few one-shot jobs. The extra time we take to compile is more than recaptured once a job goes into periodic application. Obviously a service bureau would take the opposite point of view, and correctly.

Change-over period

The Univac compiler was given a fair amount of use, but it came along at a late stage of our Univac II experience. Most of the important Univac II work predated the compiler, hence its value as a common-language device for switching runs over to the later Honeywell machines was small. When we transferred our larger Univac II programs over to the H-800, we played around briefly with some fancy ideas of how to automate the process partially, but fairly quickly concluded that it would be necessary to work directly from Univac machine coding into H-800 English language statements, and the job was done that way. One of the fancy ideas was to write a program in English language for Univac (working from the machine coding of the program), compile it for Univac and test on Univac that it produced exactly the same processing as the original run, then recompile for the H-800 using the same statements. This very quickly proved to be too cumbersome, and the approach was dropped. Deloading of Univac programs was finally done by having programmers translate Univac coding into English statements acceptable to our Honeywell Compiler.

In retrospect, the real value of the Univac compiler was in the experience we gained in producing it. Also, we stopped being frightened by tales of 100 man-years,

costs of $3,000,000 per compiler, etc., and found out that we could do this type of work ourselves, at far less cost. To paraphrase the title of a well-known movie: we "learned to stop worrying and started to love the Compiler."

The do-it-yourself approach

I mentioned that the H-800 was picked as our next generation computer at a series of meetings in 1960. This decision immediately followed our first successes with the Univac compiler, and these successes with the compiler had the effect of our tending to minimize the importance of evaluating manufacturers' software in selecting a machine. This discounting of software enabled us to concentrate on studying the hardware capabilities of the competing machines, looking at software with the attitude that if it weren't satisfactory we would do it ourselves. This, incidentally, is the real message I have for you in this talk: consider the do-it-yourself approach on software in your own installation. If the installation is small, such an effort will have to be modest, though even a small effort may bring worthwhile rewards; if the installation is large and mature, try it by all means.

When we had made a firm selection of the H-800 as our next generation machine, we had about a year's grace before the hardware would actually arrive, time to produce our own software or modify Honeywell's. As regards the English Language Compiler situation, the decision to go our own way had really been made concurrently with selecting the machine. We were pretty well satisfied with the Univac English Language Compiler, hence felt confident of our ability to extend it to the H-800. COBOL for the H-800 was still a number of years away. The FACT compiler system of Honeywell's did not, shall we say, strike a responsive chord at Metropolitan—which is somewhat of an understatement. It had a special character set, involved using what we thought was an unnecessarily complicated system of structuring data files, and featured frequent relocation in memory of a program during a run, to mention some of our misgivings. As a tool adapted for our needs, we did not give it serious consideration. I should add to these remarks by saying that some American installations have used FACT, and seem to like it.

Consideration of available software

I will return to the compiler story later, but it might be well to mention here three other aspects of H-800 software which had to be evaluated. One was the ARGUS assembly system, for converting mnemonic coding into machine coding, and bringing library routines into a completely assembled program. Another software question, though not a program, was the Honeywell convention system in regard to data tape layout, handling of signs, rules regarding the handling of parallel-running programs, etc. The third major question was the Executive Routine.

The ARGUS assembly system appeared to be in good working order, and although we did not want to use all of its various features, the ones we needed seemed satisfactory, so we decided to leave well enough alone in this area, for a few years' time at least. Much later, in December 1963, we started work on a replacement for it, and this is now coming into use in the Company. The replacement, which we call "Mercury", is much faster, and has tighter controls.

The various Honeywell programming and data tape conventions obviously needed more rigid definition at Metropolitan, and I think this will always be true of *any* manufacturer's suggested conventions on *any* large-scale machine in *any* installation. To borrow American baseball terminology, the ground rules adopted have to depend on the stadium the game is played in. At Metropolitan we have a positive passion for very complete controls, to ensure that the chance of erroneous computer results being sent to a policyholder is as close to zero as it is humanly possible to make it. Part of this feeling comes from the fact that we are closely regulated, so that our procedures are scrutinized by various states' insurance commissions. Another consideration was that however completely we had examined the H-800, only actual experience would tell us how reliable the machine really was. (In retrospect five years later we can say that the machine is remarkably reliable.) Still one other reason for more extensive controls on Metropolitan work is that we do not have the "safety in numbers" situation that many smaller companies can rely on. If the chance of some type of machine failure going undetected is estimated to be one in a million, for example, many installations may take the risk, willingly and properly, of such an occurrence once a year. We can't, because with our volume this low probability of failure may be translated into a breakdown twice a day. The end result of all these considerations was that we made some significant changes in the Honeywell data tape conventions, hence could not use the Honeywell input-output packages, hence made up our own. We also reversed the sign logic of the machine, so that "machine-positive" meant "Metropolitan-negative" and vice versa; perhaps this last change confused more programmers than it was worth, although it hasn't caused any real trouble except in people's minds, and had a positive logical value. We found out later that the FACT compiler system made the same sign logic reversal.

Executive Routine

I have mentioned the assembly system and the conventions. The third non-compiler software article we looked at was the Honeywell Executive routine. In a computer that has the capability of running more than one program at a time, the Executive Routine (sometimes called Executive Monitor) is a master program that performs the necessary function of policing the action of the other programs. It decides which sections of memory are available for new programs, keeps the human operator informed in regard to programs

beginning and ending, finds and loads new programs, etc. Honeywell's Executive routine was available in only one version, and this was geared to a much smaller configuration of memory and tape drives than Metropolitan had ordered. Honeywell's system required pre-planning of parallel running programs, in small-memory machines, where there would not be very extensive parallel running anyhow. With our larger memory and larger number of tape drives, we wanted to make extensive use of the parallel running possibilities of the machine, and instead of pre-planning series of runs, we wanted the Executive Routine to re-schedule the operation dynamically as runs finished, by examining a "waiting list" of jobs to be run when possible. With the substantial task we had already assumed of putting together a compiler, we were in no position to take on this additional programming task, but it appeared that the specifications for this type of Executive could be drawn up clearly enough so that the job could be farmed out. A committee of four specified the logic of what we wanted, and we called in an outside consultant firm to do the job. Our own Executive Routine was working and available a year later, when the machines arrived.

Temporary compiler for H-800

To return to the English Language Compiler situation: we started immediately to work on an H-800 version, that is, one to produce H-800 coding, but wisely decided to program it for the Univac II. This had several immediate advantages. First, we had a fluency with Univac II language that we knew would take some time to develop with H-800 language, and we knew how to debug Univac programs well, with six years' experience. Second, we could test it out as we went, on our own machines in our own shop, without getting into a queue for time on the few H-800 machines then working, all of which were outside our office. Third, about one-third of the job was already done, since the preliminary syntactical analysis in the working Univac compiler required few modifications to serve the same purpose in a Honeywell compiler. Of course, we knew it could not be a permanent solution, but the fact that we intended to phase out the Univac II machines only very gradually made it a very acceptable solution.

I can't resist pointing out that this decision, to implement an H-800 compiler on a Univac II, is one that only a *user* could have made, and is a good example of the advantages, sometimes unexpected, of a computer user furnishing some part of his software requirements himself. It was a slick way of getting work onto the new machines quickly. It is entirely possible that our inability to do this type of work would have set our whole time-table back a year or more. One of the many good points we considered about the H-800 was its relatively early availability, compared with some other machines we studied—this advantage might have lost its appeal if we had not been confident that our own staff would be adequately supplied with tools to exploit the advantage of an early hardware installation.

This hybrid compiler was called the "Compromise Compiler"—a misleading name perhaps, but we did not feel that the name had much importance except to identify it. We have never been very keen on tricky names in the Metropolitan computer division. The wholly H-800 compiler that eventually replaced the Compromise Compiler really has no name at all, except "The Compiler"—The Executive Routine is simply called that, or sometimes "Met-Exec." We marvel at the ingenuity that other people display in coming up with those wonderful acronyms like COBOL, ADMIRAL, FACT, SOAP, and the like. We wonder where they find the time for such research. Many acronyms have such an ultimate flavour about them that they seem to leave little room for improvement. The original UNIVAC, for example, which is now carefully preserved in the Smithsonian Institution in Washington, along with Lindbergh's airplane, had a name that meant "Universal Automatic Computer." Where can one go from there? Nor is this a peculiarly American disease: I had pangs of envy and admiration when I first heard about the great Nebula compiler in Orion; that one deserved a prize. I suppose I can sum up these parenthetical remarks by saying that we don't care what people call it, provided it works.

A few minor modifications in the compiler language were made in adapting the compiler to the H-800. These had to do with methods of identifying tapes, and some related to the matter of running under the control of an Executive routine. No changes at all affected the processing language itself, and it was fairly easy to spot and change Univac statements that required changing in order to recompile on the Compromise Compiler for the H-800.

Although the New York machines originally had 16,000-word memories, our smaller machines in San Francisco and Ottawa, installed in 1962, had 12,000 and 8,000-word memories, and in order to be able to run each other's programs, we restricted H-800 program size initially to 6,000 words of memory. This restriction had the desirable effect of permitting parallel operation to some extent, at least in New York. Where the Univac Compiler took about one hour to produce programs restricted to the size of the Univac 2000-word memory, the Compromise Compiler took from two to three hours to produce an H-800 6,000-word program—the correlation between time of compiling and memory size of the compiled program was quite close. Later, in 1963, we doubled memory size on the New York machines to 32,000 words each, and the San Francisco and Ottawa machines were increased to 16,000 words each, and at the same time we raised the permissible size of a program to 12,000 words. This of course made compiling time rather enormous on those few programs that went to very large sizes, but it occurred at a time when enough work had been transferred from Univac to Honeywell to make such time available. The "pure" Honeywell compiler (i.e. compiling runs on the H-800 for the H-800) went into operation in September of 1964,

and the whole problem of excessive Univac compiling time promptly disappeared. We had scrapped one of our Univac II machines in May of 1964 (we tried to sell it, but no one wanted it even as a gift), in order to provide floor space for an H-1800, and it was a great relief when the new compiler started operating four months later. The risk of not having enough time to compile for new trials being worked on during those four months was not a real one, since we could have bought some time outside—but this was not necessary, as it turned out.

During the years that we used the Compromise Compiler, about 500 programs were produced by it, and no other programming system had to be used for regular production type work. Programmer confidence in the system is very high, and has continued high with the introduction of the newer all-Honeywell compiler. One simple indication of this acceptance is the fact that when bugs show in test sessions the normal reaction is to examine the statements for errors first, not the coding produced. Project leaders and managerial personnel above the programmer level are completely sold on the system, and it is generally felt that the progress we have made over the last few years would not have been possible without the compiler.

Work started on the "pure" Honeywell compiler about 27 months prior to its completion in September of 1964. There was less urgency, except at the very end, and we wanted to gather up all those tidbits of hindsight that the two earlier compilers gave us, and really do a good job. *Rule:* if you want a job done well, do it the third time. After only one year of its development, the partially completed compiler started to pay large dividends, since its very complete and fast diagnostic section was working by then. This enabled us to find programmers' errors very rapidly on the H-800, then send to Univac only those programs that were clean—this simple step cut Univac compiler time requirements in half.

Language changes for the third compiler were confined to liberalizations of some prior restrictions—there were no changes that required rewriting older programs for the new system. A king-size program that requires 12,000 words of memory can be compiled in less than an hour—the average run is about 30 minutes. We are now compiling about 20–25 programs per month, not counting about twice that number per month of partial runs that reveal programmer logical errors.

Features of latest compiler

Some of the features of our latest compiler may be of interest. Half-way through a compilation an English Language *analyzer* is produced, which lists the original statements' text, and shows all cross-references between statements. This is valuable for making later program changes. Lists are also given of the various files, data nouns, storage areas and tables used in the program, showing the statement numbers of all processing steps that refer to each. If there are any errors in the program this list is also present, and the compiler quits without going further.

The Honeywell symbolic coding language, which the compiler produces, contains a provision for inserting "remarks" lines among the coding. The compiler uses this facility in two ways: first, each small section of coding, which corresponds to a clause of a statement, is preceded by remarks that give the statement and clause number, as well as the English text of the clause. Second, the entire original text of the program is converted to remarks lines that appear at the end of the program listings. Having all this information in one package helps program maintenance considerably.

Wider use possibilities

One of the suggestions made to me in connection with this paper was to make some comments on the "case for a tailor-made language." I can't do it because the system is not a tailor-made language—it is a language that is generally useful to any commercial EDP outfit that happens to use the same equipment as we do. I can think of only one verb in our compiler vocabulary that is really tailor-made, and as it is optional, it does not affect the general value of the language. In connection with our communications project it is necessary to print a bar-code at the bottom of our premium notices, so that these may later be run through an optical scanner at the District Office for transmission to the New York computer center. This consists of five rows of small vertical bars, in which each vertical combination of bar or no-bar stands for a character in the scale of sixteen. We have a verb known as *BARCODE* which sets this combination up in storage for the printer. All of the rest of the language is general, except of course that no provision is made for hardware we do not yet have.

I was also asked to comment on the matter of "commonality of language between machines." We did achieve this partially between Univac and the H-800, with the minor exceptions I mentioned above, but in the perfectly general case I think it rather impossible. One could hardly hope to achieve commonality between a machine whose main large storage was tapes and another whose main large storage was a large disc file, for example. Even between two tape machines, one may run into irreconcilable differences in the character set. I can recall one job that caused some trouble going from Univac to Honeywell where the original Univac logic depended in part on the fact that a Univac "space" or "blank" is lower than a zero in the Univac collation sequence. It happens to be larger than a zero on the H-800, and as a result a new field had to be added to the H-800 file design in order to accomplish the same processing. Will we ever have a standardized character set common to all machines? I doubt it.

Speed of object program

I mentioned earlier in this paper the preference we have at Metropolitan for speed in the generated program that comes out of the compiler, rather than speed of compiling time, and I also pointed out that I did not think this would hold true in other installations with a different

environment. There is a related subject that comes to mind: the frequently heard argument that the need for efficient coding will gradually diminish as machines get faster and faster, *ergo* one can tolerate sloppy compiler generators. I feel quite strongly that no matter how fast the internal computing speed of a machine becomes (and we seem to be approaching a limit) it will always be important to use it efficiently. Speed of processing represents money, plain and simple.—If compiler A produces programs that run twice as fast as compiler B we save expense by using compiler A.—If another machine is built three times as fast, will this justify using the compiler B approach? Of course not—Nanoseconds are just as important economically as microseconds, if there are enough of them.

Machine-code routines

One aspect of commercial EDP compilers that is near and dear to my heart concerns the "Own-coding" option. Metropolitan may possibly lay claim to having produced the only English Language Compiler that does *not* give the programmer the option to go into machine coding to do something "special," then return to generated coding. Our attitude is that if the language has been thought out and implemented properly there should never be any such need—and if one looks upon a compiler, as we do, as a device that enforces a common discipline among problem solvers, an option to use own-coding will certainly operate to defeat this discipline. Rare cases of real need arise now and then, which we solve by simply adding some language facility to the compiler. I might add that putting an own-coding option into the compiler would be rather simple—but we don't think we could live with the confusion it would produce. Incidentally, we did have some actual experience with something like this, on a different type of compiler. We had on Univac an interpretive compiler, known as the "worksheet program" which had the specialized aim of automating actuarial calculations that could be expressed and solved in terms of columns and lines of a calculation worksheet. It worked very well on things like actuarial life functions, where one expresses tabular formulae that run down one column and up another, etc. One day it occurred to me that we could put an own-coding option into it, and I did, and let out the news. It was a horrible mistake, because worksheet programs came into being that were almost incomprehensible mixtures of worksheet interpretive code and Univac machine coding. There is an evil instinct in all of us to over-complicate things—one might call it the "Chinese puzzle urge": it must be dealt with firmly.

Verbosity defended

A frequently heard criticism of English Language programming is that it is too verbose in stating a problem. Some suggest allowing one to write RD instead of READ, or A instead of ADD. A similar complaint has to do with our insistence on programmers attaching meaningful names to data fields—instead of calling them INSURANCE, COMMISSION, PREMIUM, the suggestion is to allow something short, like A1, A2, A3, on the grounds that the programmer will know what he is doing, and why write all those extra letters? There are several replies to this line of thinking:

(1) The redundancy in "ADD," as compared with "A," is a valuable error-detection device, to guard against some random key-punch error changing the meaning of a whole processing step, without immediate detection. If A stands for ADD, and S for SUBTRACT, how easy is it for a careless operator to punch one instead of the other? But mere carelessness will never give SUBTRACT instead of ADD. Very well then, but why not allow SBTRCT instead of SUBTRACT? The answer is that anyone can spell SUBTRACT, but why need one take the time to look up SBTRCT in a list of authorized abbreviations?

(2) An English Language Compiler is not merely an instrument for converting a problem-oriented language into a machine-oriented language. It is also a communication and documentation device, which allows close supervision of programmers during the development phase of a project, as well as providing for easy transfer of programs from one group of programmers to another. Where a program has been doing steady production work for a year and now needs revision, use of this type of compiler facilitates the job of refamiliarization. All of these advantages tend to drop in effectiveness if there are special symbolics used whose meaning is not readily obvious.

(3) In any event, no one programs at dictation speed, or even writing speed. The amount of time it takes to decide *what* to write is far greater than the amount of time it takes to write it.

Error detection

Another aspect of EDP compilers worth commenting on is the handling of errors—what action should the compiler take when it finds one? Our attitude is that, having found one error, the compilation process should of course go on to find any others, but having finished those parts of the compilation that deal with errors, the compiler run should stop. We take the strict viewpoint that any error, no matter how seemingly trivial, may indicate collateral errors on the part of the programmer, and we do not allow the compiler to make a presumptive change, or omit that part, and continue. The advantage of this in a large computer shop is the assurance one has that test work time (after compiling) will not be frittered away on a known-to-be defective program. The attitude that calls for forcing things through, so as to get some results at least, can be adopted in a small shop, but not in one as large as ours. I suppose one added virtue in our "all or nothing" approach is that it probably encourages more careful preparation on the programmer's part.

Conclusion

The most important thought I want to leave with you is the do-it-yourself idea. I've made this plea in talks to groups in the United States, and usually hear in reply something like "Well, if we had your money . . .," etc. But there are many installations over there whose EDP budget approaches or exceeds ours, and I think it would be fine if some of them shook off their timidity. Nor is it simply a question of money or size—there are all possible gradations of the do-it-yourself principle—even a very small outfit can profit by taking the minimum step of adopting more rigid tape conventions for example —I'm not advocating the whole hog or none at all.

Hence, to computer users I would say: try some do-it-yourself software—you may find you like it. To computer manufacturers, I would suggest that software for new systems be set up in modular fashion, so as not to handicap the user who wants to substitute his own ideas here and there.

Acknowledgements

The author records his thanks to Mr. E. L. Willey (*Prudential Assurance Company, London*) for helpful suggestions regarding the subject matter of this paper, and arrangements for the meeting at Northampton College of Advanced Technology, London.

Reference

FINELLI, JOHN J. (1960). "Development of EDP Units," *The Computer Bulletin*, June 1960, Vol. 4, p. 10.

Correspondence

To the Editor,
The Computer Journal.

Character recognition

Sir,

I would like to comment on the article, "Character Recognition," by F. H. Sharman in your July issue.

Firstly, I consider that this paper has done a valuable service in bringing out U.K. views on Character Recognition, that have been dormant for some time, and that these views can now fruitfully be discussed. However, these views, as presented in the paper may be initially biased due to the fact that organizations who expressed "no use foreseen" were nevertheless included in the analysis of answers to the subsequent questions. For example, Table 2, Section *b* shows 7 out of 15 organizations having "no use foreseen," but Tables 4, Section *b*, 5 Section *b*, and 7 Section *b* show many more than 8 organizations' answers being analyzed.

The authors have clearly met misunderstanding of errors and rejections, and their clear-cut distinction in Section 10 is to be applauded. On the other hand, I feel they trod on dangerous ground, in acting as judges, by declaring that "The replies which indicated a rejection rate similar to that experienced with punched cards was probably the most realistic." By use of an intelligent total system with context correction within a document, or within a batch, this may be achievable; but considering a document reader by itself, faced with marginal quality printing, then it would be ambitious to suppose that it could match a card reader.

Regarding errors, the authors were themselves not wholly consistent when they stated in Section 3, that "steps are taken to eliminate these (input errors) with cards and paper tape, so techniques have been devised to ensure accuracy in Character Recognition." Later in Section 10, they state that the requirement for no misreads is a "state of bliss which has never been achieved in any form of input."

This goes to show how precise one's wording must be when writing on this subject . . . unless I am alone in my interpretation of "eliminate".

Certainly any form of input mechanism has a finite error rate and be it 10^{-5} or 10^{-7} it is *never* zero; it is wishful thinking to assume that integrated circuits, self adaptive systems or what have you, will change this state of affairs.

The summary is quite fair, particularly in its final paragraph. The only points that I would dispute in it relate to proportionally spaced and easily read fonts. For the former it is my view that people liking proportionally spaced fonts really applaud the print quality since these fonts are almost, if not always, used on electric typewriters with good class ribbons. Further, if one is looking for an easily-read font the ECMA B font is a very good candidate for this.

Yours faithfully,

J. BAULDREAY.

Rydal,
Heathfield,
Royston, Herts.
21 September, 1965.