BIT 6 (1966), 58–65

ON-LINE USER LANGUAGES*

JOSEPH WEIZENBAUM

Abstract.

The principal differences between on-line and off-line user languages are discussed in some detail. The characteristic features of the OPL programming language are mentioned together with experiences of Project MAC, pointing at increased efficiency in the use of computers.

The title of this talk** suggests that there is some difference between on-line and off-line user languages. The fact that such a talk was invited at all at this time suggests further that there is a renewed interest in that distinction. I say "renewed" because our experience with computer languages began with on-line systems. Those of us who are now privileged to have on-line access to large scale computers often have a distinct *Deja vu*-feeling; we have been there before. Of course we have. It was in those far away days when the only way to communicate with computers was by means of the direct coupled input/output-devices euphemistically called "consoles", often consisting of arrays of switches and buttons of such bewildering complexity as to provide a challenge to a modern airplane pilot. A little later, some computers were designed by people who had actually used earlier models themselves. That accounts for the appearance of typewriters on some early machines. Many of the modern small machines are direct descendants of these early models, differing from them only in that they have core memories in place of drums or mercury delay lines and in that they are very much cheaper. The early machines were operated in an on-line mode because techniques for efficient batch processing were not yet developed, today's small machines because their economics are not prohibitive. Contrariwise, most large scale computer systems are *not* operated in an on-line mode because the

* Work reported herein was supported by Project MAC, and M. I. T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research contract NONR-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

** The paper was first given as a talk at a symposium on on-line computing organized by the Extension Service of the University of California at Los Angeles in the spring of 1965.

economics of such operation *are* prohibitive. I shall return to the economic question below. For the moment, I wish merely to implant the idea that, were it not for economic constraints, most computer users—certainly most programmers—would prefer to deal with a computer on-line.

It is sometimes suggested that a very large fraction of computer time is used in running programs merely to find out why they don't work. Whatever the statistics might be, it is certainly true that debugging of programs is one of the chief preoccupations of computer users. But it ought to be remembered that a program is somebody's strategy for solving some problem no matter whether that program has bugs in it or not. The debugging of a program is another problem. It is such a common one that sets of techniques for attacking it emerge out of the common experience.

What is really going on when a programmer is engaged in debugging his code? In effect he is doing science. By this I mean that he is interrogating some ill understood aspect of nature. He forms hypotheses based on his current understanding, designs and applies tests for verifying them, intellectually analyzes the outcomes of his experiments, modifies his hypotheses, and so on. When he finally understands the nature of a particular bug, he modifies his code to remove the difficulty. The crucial difference between this kind of activity and that of writing programs is that debugging is an exploration of a solution space with the aid of a computer while the latter is merely the encoding of solutions already arrived at by other means! We all know how absolutely necessary the computer is to the debugging process. It is difficult to the point of impossibility to diagnose an ailing program by pure reason alone. This then explains why the first language systems which can truly be classified as "on-line" were those, like DDT for the PDP-1, designed as debugging aids.

What I am really trying to emphasize by means of this argument is the principal operational advantage to being on-line with a computer, namely that that mode of operation permits the computer-aided exploration of solution spaces as opposed to the mere exercise of programs representing already encoded solutions. It seems to me that the facilitation of this exploratory use of computers is the entire aim and direction of all work in computer language development and of all computer systems design not motivated by pure data reduction requirements. It follows then that on-line computer languages should be looked upon from this point of view and judged on the basis of criteria derived accordingly.

On-line access to large scale computer systems has now been made possible as a consequence of the development of sophisticated time sharing systems such as the ones of which Project MAC at the Massachu-

setts Institute of Technology is an example. While it is no doubt appropriate to list the various on-line languages which have been developed within Project MAC and to catalogue their characteristics, it is perhaps more useful to first call attention to an important difference between simply having on-line access to a large computer system and having on-line access to a large time shared computer system. A user having his own IBM 7094 has access to whatever software happens to come with his machine plus all he manages to add to the library as a consequence of his own efforts. A time shared system, on the other hand, is constantly enriched by the combined efforts of *all* its participants. Our experience with Project MAC serves to underline the importance of this distinction with very great force. Of course, an extensive executive system is required in order to make programs developed by anyone available to everyone and to honor constraints imposed by program and file protection conventions at the same time. However, the operation of a time shared computer system demands a complex executive program in any case; not much more is needed to meet this somewhat expanded goal.

The above observations are relevant to the discussion of on-line user languages in that the various system commands which are, so to speak, at the fingertips of the user of the time shared system constitute a kind of on-line language in themselves. As it happens, no one has taken the set of such commands within Project MAC and codified them in, say, Backus Normal Form. That may, in fact, not be possible. If it isn't, then this must be because this set is very much *ad hoc*. But, lest this be taken as a criticism, let me add quickly that it follows from the very way in which such commands are added that no strict advance plans can be made. It is not possible to predict, after all, when some user may come up with something of general interest and utility.

For the present purpose, we may look upon the time sharing executive as an elaborate interpretive program. Most interpreters have some mechanism for getting at the "next" instruction. There may be a pseudo-program counter, for example. The time sharing executive gets its "next" instruction from the input buffer associated with a currently active program whenever that program has ascended, so to speak, to the system level. The set of instructions which it will understand and to which it will respond constitutes the set of "system commands" alluded to above. The "interpreter" itself imposes some syntactic constraints on the sequences of commands it will accept (e.g. certain sequences of commands are ungrammatical). The view of the time sharing executive as an on-line language structure is reinforced when it is noted that it is possible (within the MAC system) to file chains of system commands, i.e. in effect to write

a program in the vocabulary provided by the set of system commands, such that when that program is subsequently executed, the effect is as if each command were typed in separately and in sequence after the previous command has been obeyed. One way of making a new program, i.e. one written by one of the MAC users, publicly available to all MAC users is to modify the interpreter to enable it to respond to a new instruction, namely that which loads that new program into core when called upon and prepares it for execution.

There are, however, certain languages within the MAC system which may be characterized as "on-line languages" in a more conventional sense. These all share the property that they are interpreters. (I use the word "interpreter" somewhat guardedly for I don't believe that a very precise distinction between interpretation and compilation can be made—nor do I believe that such dichotomies are useful.) Clearly an on-line language within a time sharing system should have the property that its user be able to engage the machine in more or less intimate conversation, i.e. to exchange messages with it on a give and take basis and with humanly tolerable frequency. This means that the user will write generally very short programs leading to intermediate results on the basis of which he then decides on his next course of action. It is fairly obvious, and indeed it proves to be so in practice, that this mode of operating a program contributes very greatly to the kind of exploratory use of the computer mentioned earlier. An important consequence of such operating practice is that the programmer does not have to anticipate every possible eventuality and account for it somehow in his program (even if only to provide an error exit).

Any already existing interpretive system (e.g. LISP [1]) which normally expects its inputs in the form of sequences of cards and delivers its outputs to either tape or printers can easily be modified to look for its inputs from the typewriter console and deliver its outputs to the same instrument. One such programming system is an interpretive version of SLIP [2] augmented by arithmetic and control functions. This system is called OPL (*O*n Line *P*rogramming *L*anguage [3]). Up to a point, OPL's architecture is a prototype for a number of other such systems operational in MAC.

The basic input to OPL is a list structure. Generally speaking, an input list structure contains a number of expressions in functional notation, for example "LOG(SQRT(X))", separated by "$". All the functional operators except the four basic arithmetic operators and "=" are prefix operators, the exceptions are infix operators. Since, in more or less standard list notation a "(" denotes the beginning of a list or sublist and a

")" the end of one such, the input list structure is already in the form of a tree by the time it needs to be interpreted by the OPL machinery. OPL is itself written in SLIP, and SLIP list processing operators are used to administer the entire interpretive process. This means that a large part of the SLIP library must be in core during the interpretive cycle. Since this is necessary in any case, the same SLIP functions used by OPL internally may as well be made available to the OPL user explicitly. OPL therefore contains a table (essentially a large transfer vector) which associates with the name of each available SLIP function the entry point to that function. The same table similarly gives the entry points to the non-SLIP functions which round out the OPL system. The final result is a quite general purpose programming language of a power roughly equivalent to that of LISP. It is, however, considerably more mnemonic than the latter. Since it was designed to be an on-line language it has certain features which merely transformed interpreters arising out of different motivations do not (but could easily be made to) have. One example of such a feature is that a previously undefined function may be called in a program (no matter how deeply nested). When the call is encountered, a message to the effect that an undefined operation has been encountered is typed out and the program held in abeyance until the programmer enters a program defining that function. From then on (and for the purposes of that program) the newly defined function behaves just as any other built in or previously defined one.

OPL programs and their associated data structures may, of course, be stored on the disk. The OPL user may therefore build up very complex data structures over a long period of time, experiment with them on line whenever he wishes, and freeze them in intermediate states for subsequent exploration. A simulation of the effects of various organizations of a business firm may, for example, start with a quite simply developed tree representation of a small firm. Programs which compute differing budgeting strategies for such a firm may be exercised on line, and the most interesting ones saved on the disk file. As experience with the behavior of the model accumulates, the firm can gradually be grown both in size and complexity. The model maker may sit at his typewriter for hours, pushing the model around, modifying it, testing its sensitivity to this or that. In the process he is of course spending most of his time thinking—in fact he is using (and being charged for) very little machine time. At some point he will store his program in its then current state to continue manipulating it a few hours, days, or even weeks later.

OPL is, as I pointed out, a general purpose on-line language. There exists a number of special purpose languages within MAC which have

roughly the same structure as OPL. One of these is COGO (Coordinate Geometry Program) which was developed and is used by the M.I.T. Civil Engineering Department. It is similar to OPL in that it too uses essentially the same transfer vector philosophy. The functions which may be called have all been previously defined and compiled in either FORTRAN or MAD. COGO is used by students in the Civil Engineering Department for such purposes as laying out highway interchanges and the like. The programmer defines points in a two dimensional space and asks for them to be connected by various curves and straight lines. He then interrogates the system about the consequences of such connections, e.g. the areas determined by various enclosed surfaces. An important property of COGO and other programs in the same class is that the student needs to know literally nothing about the structure of the COGO program itself nor indeed about programming in general. STRESS is a similar program which deals with the stress analysis of structures under loads. The principal reason the users of these programs need know nothing about programming is that both COGO and STRESS are essentially self teaching. By this I mean that they not only respond to the stimuli provided by their users in the sense of yielding intermediate results, but give directions for their proper use. A particularly strong example of such a program is one developed by Hansen and Pyle for the design of nuclear reactors [4]. This program not only asks for its relevant parameters by name e.g. "What fuel do you intend to use?"—but also comments on the relative appropriateness of the response under certain condition—e.g. "That's rather large"—and gives the user a chance to change his mind before going into its calculation phase.

OPL, COGO, and STRESS are each programs which, while being on-line, are still conventional in the sense that their users specify procedures in terms which are generally recognizable as program steps and produce results which other programs might well also produce. A radically different class of on-line programs is represented by the "ed" and "typset" programs available within the MAC system. These are both editing programs. They differ from one another mainly in that the "ed" is designed to process computer code (in whatever language) while "typset" is a general text editor. I will discuss only the latter.

Typset operates in either the "input" or the "edit" mode. In the former the user types in whatever text he pleases, using all characters available on either the IBM 1050 or the Teletype keyboard (with the exception of two escape characters one of which is used to delete an entire line and the other a single character—both may be changed at will). In the edit mode the text is edited entirely by context. There are no concepts such

as line numbers nor any others requiring the user to remember the appearance of the original text. In the edit mode, paragraphs may be interchanged, new text inserted between pairs of words or characters, strings of characters deleted, and so on. Text which has been input via the typset program is ultimately stored on the disk and may be output on any kind of paper (e.g. ditto masters) at any time. Since it does reside on the disk until wilfully deleted therefrom, it may be re-edited repeatedly days, weeks, or months after its original composition.

It seems to me that the typset and ed programs are significant for two quite disjoint reasons. One is that they provide examples of programs which anyone may use to very good advantage—anyone, regardless of how little or much computer programming he has behind him. Even I have been able to turn out papers with lines properly centered, left and right margins justified, and the proper spacing between paragraphs. Perhaps of more immediate importance is the fact that both these programs were written, debugged, and put into general use within a few weeks of their inception. This, while of credit to their authors, certainly also serves as a comment on the utility of a time sharing system and the on-line program writing and debugging facilities it provides to each of its users.

I shall now return to the question of economics which I postponed earlier. As I have already indicated, I believe the economic issues related to the on-line use of computers must be separated according to whether one is speaking of small computers operated from their directly connected consoles or large time-shared computer systems. The more challenging and interesting questions certainly lie in the latter category. Visitors to Project MAC are constantly asking how much machine time a problem they solved on a batch processing system would require on the MAC system to come to the same state of solution. I suppose they want the answer in terms of numbers of machine cycles or some related measure. Perhaps these questions can be answered with precision in some cases. Whether such answers would be impressive would depend, of course, on the efficiency of the batch processing system with which the MAC system is being compared. Batch processing also entails overhead. However, I don't believe such questions to be very meaningful—no more so, for example, than asking for the average floating point add time of a machine like the STRETCH leads to any insight into the overall effectiveness of such a machine in relation to specific problem classes. It must be remembered that economics deals with the allocation of scarce resources. To consider the number of machine cycles required to solve a particular problem to be the principal measure of

effectiveness of a system assumes that the scarce resource to be conserved is machine time. It is not. In any case, one would probably not take the same code as that which was generated for the batch processor and run it in the time-shared environment. For, in the batch processing environment the programmer knew that access to the machine was limited to a few shots a day. He wrote his code in anticipation of every conceivable event, asked for large dumps, interleaved the operationally significant portions of his program with elaborate diagnostics, and so on. None of these precautionary measures are necessary with on-line operation. Programs are therefore shorter and run correspondingly faster.

The scarce resource which is being conserved in the on-line mode of computer operation is the energy and time of people. Our experience with Project MAC has taught us that the exploratory use of computers, such as has by now become habitual with us, serves to amplify the effectiveness of people in dramatically visible ways. Just as the introduction of the computer itself made possible attacks on problems which simply could not be attempted earlier, so we find that our mode of operation encourages people to search their own problems more deeply, to be dissatisfied with sloppy solutions which might have passed earlier because "after all, they work". In short, we are beginning to see the use of the computer as a real and significant assistant to human beings engaged in problem solving. Who is to say what economic values accrue to an institution when creative people complete their tasks in days instead of weeks or when a problem which could previously not be attacked at all is now solved?

REFERENCES

1. McCarthy, J., et al., *LISP 1.5 Programmer's Manual*, M. I. T. Press, Cambridge, Mass., 1963.
2. Weizenbaum, J., *Symmetric List Processor*, Comm. of the ACM, Vol. 6, no. 9, Sept. 1963, p. 524–544.
3. Weizenbaum, J., *OPL-I: An Open-Ended Programming System within CTSS*, Technical Report MAC-TR-7, M. I. T. 1964.
4. Pyle, I. C., *Data Input by Question and Answer*, Comm. of the ACM, Vol. 8, no. 4, April 1965, p. 223–226.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS
U.S.A.