MAC-TR-7

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

OPL-I
AN OPEN ENDED PROGRAMMING SYSTEM WITHIN CTSS

by J. Weizenbaum

April 30, 1964

ABSTRACT

OPL-I, an incremental programming system presently operating with CTSS, permits the user to augment both his program and his data base during widely separated successive sessions at his terminal. Facilities are provided which make it possible for the user to operate on his already established data base both by means of built-in operators and in terms of operators (functions) which the user has previously defined in the language of the system. Underlying the system is a powerful list processing scheme imbedded in FORTRAN (SLIP). The machinery of this fundamental language drives the system and is also largely available to the user. The data base generated by the user is therefore a set of list structures (trees), and most of the operators available to him are list processing operators. Data structures with considerably complex interrelational properties may therefore be treated quite directly.

"Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

This empty page was substituted for a blank page in the original document.

A time-shared computer system such as at M.I.T.'s Project MAC[(1)] is rich in opportunities to attack problems in new ways. From the user's point of view, his typewriter (connected to the computer) is very much like the relatively simple control consoles of the computers of long ago. The speed of response to the signals he sends to the computer tends to confirm the illusion that he has a computer all to himself and that he has been thrown back in time to those long gone days when console debugging was de rigueur. It is probably true that the old timers gave up this mode of computer operation most reluctantly and only under the unchallengable economic realities of large, fast computers. The sudden reversal of events is therefore greeted with the kind of pleasure associated with the rejuvenation of an almost forgotten romance. And, just as it would be a mistake to believe that the rediscovered object of one's long ago affection has remained unchanged over the years, so in this context is it wrong to gloss over what has happened to computers since one last sat at one's own console. The most directly influential changes have been the increases in memory capacities, both core and bulk, and the development of high level computer languages. Indeed, were it not for disc and drum storage, time-sharing would be impossible for there would be no efficient means for swapping programs in core or giving active programs rapid access to previously stored files. High level languages are almost certainly required to write the complex executive programs which form the basis for any time-sharing system as well as to make these programs amenable to maintenance and change. Of course, the availability of high level languages opens the door to the system to users whose main concern is with their problems and not with the computer per se.

The most obvious, and in a sense most primitive, effect the user of a time-sharing console notices is, of course, an impressive reduction in turn-around-time *vis-a vis* batch processing. He types in, say, a FORTRAN program, compiles it and is given his diagnostics within a very few, if not within fractions of, minutes. He can then place his missing parentheses, relabel his mislabeled statements, or whatever, and recompile. This would obviously not be possible if the compiler itself were not accessible in the form of a previously stored file. The psychological effect of being able to recompile several times in one sitting must be experienced to be appreciated.

But the most suggestive aspect of the freedom with which one may compile, repair, and recompile is not in the mere reduction of turn-around-time. It is rather in that this type of man-computer message exchange (man submits program--computer points out bugs--man submits revised program, etc.) is an (albeit primitive) example of a qualitatively new realization of man-machine dialogue. The whole point of time-sharing is to enlarge the opportunities for carrying out truly significant man-machine dialogues--not to merely reduce turn-around-time. The goal is to give to the computer those tasks which it can best do and leave to man that which requires (or seems to require) his judgement. It is to be expected that in many problem areas the computer will begin to help man by doing only the most obviously mechanical parts of his problem but that, as the man-machine dialogue extends over a long period of time, more and more of the previously fuzzy issues over which man retained authority will become clear and finally be turned over to the computer. There will, in other words, come into being heuristic computer programs in which the heuristics themselves will be products of man's computer experience, of his deepening understanding of his problem as a direct consequence of solving it in partnership with a computer.

Whatever the computer language techniques which may be required to compose such emergent programs may finally turn out to be, they are certainly not those which have proved effective for the batch processing discipline to which we have all become accustomed. That discipline requires a user to anticipate every possible eventuality in the sense that for every such eventuality a program dealing with it has to exist at load time. In an important sense then, it may be said that batch processing requires the programmer to have a fairly complete idea of the solution of his problem before he can even begin to appeal to the computer. The computation is merely the evaluation of certain parameters identified by the programmer in advance.

In order to exploit the opportunity for programmer interaction with an ongoing computer program which the MAC time-sharing system provides, the idea of presenting the computer with a completely specified (in a sense, black box like) program must be abandoned in favor of a technique which permits the user sitting at his console to initiate computations which carry only to a point of uncertainty, at which point then the user can make further programming decisions based on judgements exercised in the light of results so far attained.

The basic purpose of OPL-I is therefore to permit the user to build programs and data bases incrementally and over periods of time during which there will be long intervals of no user-computer interaction at all. The SAVE and RESUME features of the MAC system are essential to this end. The first of these permits a user to cause the entire state of his program to be stored on the disk files under a file name chosen by himself. The second causes a saved file to be retrieved from the disks in such a way that, even though weeks may have elapsed between the SAVE and RESUME operations, the program which was underway at the time of the SAVE is continued as if no interruption had occurred.

Experience has shown that one of the most powerful data storage schemes relevant to present computer organizations is the <u>list structure</u>. Its chief advantage over "conventional" storage methods is that the very storage regime itself (as opposed to programs dealing with the stored data) permits the recording of complex interrelationships among the data. List structure representations of programs also yield considerable economies in programs to process such programs. OPL-I is therefore fundamentally a list processor very much in the spirit of LISP[2] and (less so) of IPL-V.[3] The executive program which drives OPL-I is itself written in a FORTRAN based list processor, SLIP,[4] all of the machinery of which is available to the OPL-I programmer.

In OPL-I the programmer enters program segments and data during any given session at his console, executes some of his program steps, thereby perhaps modifying his data set, and finally quits by saving his accumulated program, frozen, so to speak, at its last step. He may resume his program at any time thereafter, save again, and so on.

Segments of programs operating under these conditions fall into two classes: those which are executed repeatedly, i.e. essentially subroutines and those which are exercised only once. Were the latter type accumulated in storage along with the former, computer memory would soon be filled with useless material. It would then become the user's task to purge his program of such material. This would place an unacceptable bookkeeping burden on his shoulders. In OPL-I, therefore, almost all program segments are deleted as soon as their execution is completed and the space required for their storage returned to a pool of generally available space. The exception to this rule is invoked if the entering of a program segment is preceded by the word "DEFINE". Such a program segment is treated as a procedure, is permanently stored, and may be called upon at any future time. The ability to so define and store subroutines means, of course, that the user has a system which he may mold and modify to his own ends. It is also important to remember that although program segments which are not identified as procedures are thrown away upon being executed (hence making programs of unlimited lengths possible), the consequences on data of the execution of such programs are stored.

An example may serve to illuminate the point. If the programmer writes:

((A = 1.5) (B = 2.1) (C = POWER(A,B)))

then, upon pushing carriage return on his typewriter, this small program segment is executed and finally thrown away. However, the data A, B and C will have been placed in memory with their proper values, C being $(1.5)^{2.1}$, and may be operated upon by subsequent program segments. If on the other hand, the programmer writes:

```
(DEFINE)
(MEAN(L)
            (S = SEQRDR(L))  (SUM = 0.0)  ((COUNT = 0.0)
BEGIN       (C = SEQLR(S,F))
            IF(F)MORE,MORE,DONE
MORE        (COUNT = (COUNT + 1.0))
            (SUM = (SUM + C))  GOTO BEGIN
DONE        ( (SUM/COUNT))  )
```

then he will have stored a procedure which, given a list of numbers, will compute the mean of those numbers and deliver that result as its value. However, the completion of the input of the above program segment--as signalled by the carriage return following the typing of the last right parenthesis--does not itself cause the procedure to be fired. Execution of a statement of the form (for example)

```
(X = MEAN(SET))
```

will fire that procedure.

It is beyond the scope of this presentation to give a complete catalogue of all the built in functions, control statements, input/output and diagnostic facilities of OPL-I. Suffice it to assert that, viewed as a language, OPL-I is of a character quite similar to the LISP program mode and of about equivalent power.

The importance of the fact that OPL-I is fundamentally a list processor operating in the incremental data and program aquisition mode already discussed is that this combination makes possible the experimental manipulation of complex data structures and their interrelations. List structures are particularly appropriate because sublists of lists may be easily and naturally interpreted as subparts of whatever the list stands for. Furthermore, lists have no inherent dimensionality, i.e. their size may vary drastically during program execution without causing programming difficulties. It is also possible to attach so called "description lists" to lists, i.e. storage devices which contain information about properties of the object their host list is supposed to represent. Previously existing list processors had all the power which such an immensely flexible data organization yields. However, programs written in these systems still had to be complete specifications of a single computational procedure--however long and complex--and did not therefore permit direct human observation of tentative results nor immediate human redirection of the ongoing computational process in the light of such results.

A simple example of some of the above points is the following: Suppose an organization is described in a standard organization chart format, i.e. the top level of management is the head of the tree with as many branches flowing from it as there are sublevels (say divisions) reporting to it. Each division is again a "node" of a tree with branches flowing from it. In this way an arbitrarily large and complex network can be represented. Within OPL-I each such node is actually the head of a list which is a sublist of the higher order node from which it flows, i.e. to which the suborganization so represented reports. The top level of management is (appropriately) the "main" list. Each of these lists may have description lists attached to it which contain arbitrary information about the represented component, e.g. the name of the component manager, the size of the budget and of the manpower pool, data on last year's performance, etc. The organization structure is therefore known by the very way it is represented in computer storage, not by means of programs which define it.

It is now easy to write programs which make all sorts of computations on this data base. For example, one program might allocate the budget of each higher level as a function of the requirements of lower levels reporting to it. It is now a trivial matter to radically reorganize the entire structure on an experimental basis and to see what effect such reorganization has on the over-all budget, inventory costs, etc. This reorganization can easily include the aquisition of new or the deletion of existing subdivisions. All programs which were written to perform computations on the original organization remain invarient with respect to any such reorganizations. In any event, the new results can be seen directly and further computation based on the insights so gained initiated immediately or very much later. If the reorganization is to stand, then the program which contains that representation can be saved and will, of course, be the already updated representation required next time that program is resumed.

A facility soon to be available to time-sharing users will permit a number of people sitting at separate consoles remote from one another (and, of course, from the computer) to interact with a single program. This points the way to the next most obvious powerful extension of OPL-I. In the multiple user mode it will become possible to simulate group processes (e.g. business games and behavioral science experiments) with utmost realism. For them the consequences of any single individual's decisions will immediately modify the model either in terms of its data base or of its very program.