that drives the syntax table can be written by one person in a couple of weeks, and the syntax table itself goes together very well.

Schwartz: The part of the problem which you mentioned as important for these higher-level languages is getting the algorithm down. It is my opinion in viewing these real-time systems that this particular phase of the job is actually quite small, no matter what the language is. Somehow, when the specifications are clear, which they normally are not, the getting of the program written is not the hard problem. Debugging is the hard part; but still, the part where the language is the most important is the area of change of these programs. If you can design these kinds of languages to permit rapid program change with faily complex changes in data, that's the key to the problem, rather than just permitting someone to write coding a little faster.

Opler: That's a very good point. You have your system almost fully developed when they announce either a slight change in hardware, a major development, or a new external unit that's to be hooked up to your unit. What Jules [Schwartz] is saying is that it would be manna from heaven if you could change a few statements and recompile your real-time program.

Steel: The production of real-time systems is a real-time problem.

Schneider: We have had a few comments that real-time is not the same as online, but there is considerable overlap. It seems that the hardest problems [those mentioned] came up in the areas of online, real-time applications. I am wondering what might be needed in the areas which are either online but not real-time, or real-time but not online?

Opler: First, where real time and not online is concerned, it generally turns out that, once you've cut yourself away from reality (in the online sense), all you're doing in the computer is mirroring real time, rather than interacting with real time. Therefore, the problem is that of getting the object program to do its execution in a time equivalent to real time. Now it turns out that if real time is slow, time is no problem; if real time is moderately fast time, it's no problem; if real time is extremely fast time, it may turn out that the compiler will be required to be constructed in such a way that all emphasis is pushed on producing a very, very fast program even if it takes several hours to compile the program. Now, if you have online computing that is not real time, the problem turns out to be less constrained.

Steel: This little piece of code, which is in no known syntax, is an example: *Do Job X, for i = 1, by steps of 1, to n within Time = 3n + 27 in some units*, where there is a requirement explicitly placed in the language to demand that the computation be done within a certain time. Do you know if there is any known procedure for dealing with this type of problem?

Gosden: Is there an **else** statement on the end?

Sammet: One of the problems in creating situations and systems of this kind is the problem of allocating resources and knowing what, in fact, is important and what is not, and how long things can be delayed. If I can, I should supply to the system a statement that says, "I want the following things going, and in the following period of time," which might be one minute, one hour, or so. I think this would help tremendously.

Opler: I really must apologize because I neglected a whole class of work that has been done. There are in existence probably half a dozen systems which enrich the FORTRAN language by adding to it a series of terms associated with real-time or process-control systems. These systems have been developed, but they're fairly primitive, I believe, along the lines of what is needed to compile the types of programs to cope with these systems. These systems have been developed in particular with the view of processing control. Statements in them facilitate analog-digital conversion and establish a time. It may be that some of these have a statement of the sort: "The following must be done every three seconds."

Online Programming

Jules I. Schwartz

System Development Corporation, Santa Monica, California

When the transition has been made from offline to online programming, there are a number of changes in the working conditions noted. These changes in the environment make necessary corresponding changes in the processes related to producing and checking out programs. In the main, it is not the programming language itself which must be changed to provide a facility for the online user; it is the system surrounding the programming language. In this paper the online environment and its effect on programming are discussed.

Introduction

One might suspect that there should be considerable difference between programming languages intended for users having access to online devices while producing or executing programs and languages used in a strictly offline environment. This is not true. It is true that some programming languages that are intended for online use differ to a considerable extent from languages of the same family intended for offline use [1]. However, these differences probably arose from deficiencies noted in the offline language while it was being used online, or because the online language was easier to experiment with than the offline one. In either case, language characteristics used in online programming languages generally apply to the offline languages as well.

Of course, due to the nature of online processing, there are languages that are associated almost exclusively with online use. For example, Joss [2] and the Culler-Fried system [3] are languages that were intended for use at a console. However, these are not programming languages;

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

thus their major language characteristics reflect an attempt to assist the nonprogrammer in solving his problem, rather than reflecting any innate online language difference for programming.

There is no question that with increasing use of online devices and systems, there will be a substantial increase in so-called user-oriented or application languages. Since most scientific or investigative applications require considerable interaction, the lack of facilities to interact has hampered to a great extent the development of languages oriented to the nonprogrammer. Thus in most instances, better programming languages have been developed so that the professional programmer can better assist the applications-oriented user with his tasks. The major language development efforts have been in the area of languages which are quite general in nature, and also oriented towards the computing—rather than the applications—specialist. These programming languages are the ones where little difference should be reflected in the transition from offline to online operation. Although the increase in online systems will create inroads into the need for the professional programmer and his languages, there will continue to be interest in these for a considerable period of time.

There are differences between *using* a programming language online and using one offline. However, these differences are caused by, and are reflected primarily in, the system surrounding the programming language. The physical surroundings—consoles, hard copy output, displays—affect, to a considerable extent, the way a person deals with a computer. The fact that a computer responds at a different rate when used online also affects the requirements of the user and consequently the system being used. Thus, the differences between online and offline programming lies in the *entire programming process*—from the time the requirements of a program are decided upon to the time it is checked out. The programming language used is a small part of this process.

Differences Between Online and Offline Access

Online. When users use a computer online, they have terminals with which they can have an almost constant dialogue with or about a program or programs throughout their stay at the console.

This dialogue can take place before, during and after the instruction execution of the program. In an offline system, of course, the "dialogue" (if it can be called that) has a much lower frequency, can be more extensive, and takes place from user to program prior to execution and from program to user after execution with no exchange during execution.

Time-sharing. Since the user's part of any online dialogue normally requires some seconds or minutes of preparation, during which time the computer has no service to perform for the user, a need exists on most computers to occupy this idle time. The most common technique is time-sharing, where the user "shares" the computer with a number of other online users or with work being performed for offline users.

This "sharing" of the computer, incidentally, does tend to moderate to some extent the previous statement that there is little difference between on and offline programming languages. Since most online languages are used in a time-shared environment, certain operations are frequently necessary in order to get or release access to the system's facilities. Examples of these can be seen in Dennis and Van Horn [11]. The additional operations exist, however, because of the requirement to share the facility. They are not due directly to the "onlineness" of the language's use.

Net effects. With the combination of time-sharing and online use of a computer—and, in rare cases, sole use of a computer online—a user can run the computer at his own pace, getting reasonably rapid response from the computer, and inputting when he feels ready. Thus, he can direct the run without concern for optimum computer utilization, in the sense that the price he pays for his own slowness should be reasonably low.

Meaning of Online Utilization to Users

The "conversational" environment available to an online user implies certain conditions that are somewhat different from those of an offline environment. Some of these are as follows:

A complete plan isn't necessary. The concept of *directed* computer utilization, techniques of trial and error, and those solutions that require human assistance for convergence are all possible with this mode of operation. Thus in program debugging, one need not fear that one small omission or error in the procedure will cause a lost run as would happen in an offline environment.

Program errors aren't catastrophic. In a good online system, the presence of a number of errors in a program should not cause any serious problems. In fact, online discovery and repair of program errors is one of the areas where online computing seems to be very strong. In an offline system, each error might cause hours' or days' delay in checkout.

The online user gets low-volume output. Online input-output devices are generally quite slow. With the exception of display scopes which are not now in general use—the amount of output that can be presented in a given period of time is considerably less than with devices associated with offline programming. Even if these terminals were much faster, it is unlikely that an online user could make much use of the speed, since he normally does not have enough time to absorb much output.

A user enters his own inputs. Unlike in the offline computing center, the user normally enters inputs directly

into the computer. In the offline center, he has it prepared by others, and the inputs are entered by other people or input directly by equipment. In online operation, the user generally enters commands or programs by keyboard devices, which are not intended for rapid or high-volume input. Thus, the means of expression must be fairly concise to accomplish a maximum and minimize input errors.

An online user is occupying his own time to run programs. Unlike in offline processing, the online user generally is spending his own time during the entire programming process. Some people are not too happy with this aspect. They would like to deliver their jobs and retire to their office or homes until the job is run. These people seem to be in the minority, since most feel their time is worth the gains of online use. In either case however, users are often impatient with minor problems presented by an online system. People become quite annoyed when some system inconsistency, error or malfunction causes them to lose time when they are at a terminal. Big problems or malfunctions that cause the loss of several hours at a console are catastrophic, causing even the most ardent devotees to lose enthusiasm.

The Programming Process

Since there is some direct effect on the techniques of programming, given the general considerations discussed previously, some observations on these can now be made.

Design and coding. In several systems in existence today (e.g., MAC [4], SDC [5]), many of the currently operating programs formerly ran in an offline environment and now perform quite effectively in an online environment. Aside from the addition, to most of them, of considerable human interaction, these programs are generally the same as designed for their original offline environment. Thus it appears that the design and coding of programs are not affected considerably by the fact they are to be run in an online environment.

One difference may be the fact that the online programmer is more likely to program in smaller modules and build up larger programs by connecting a number of routines. This is because it is easier to enter small routines, and checkout of small routines online is usually rapid because the turnaround time is not great enough to seriously outweigh the time spent in finding few or no errors. Of course, it is also easier to type small routines than long ones at one sitting.

It is at this stage that the programming language plays a part. One of the most important properties of an online language is the ability to easily state interactive requirements (e.g., communications with keyboard and display devices). The language should provide concise and powerful statements that enable a dialogue between user and program to be entered and changed rapidly. In general, the need for conciseness of expression is a valuable one for online languages, although this does not mean that this property is a bad one for offline languages.

Entering and modifying code. There are methods for entering code on an online system that are identical to those used offline. The code can be keypunched, then put on magnetic tape and processed from tape.

Techniques that normally exist for online systems permit routines to be entered via keyboard, then merged with other routines, at the time of entering, compilation, loading or execution. Usually, the lines of programs are assigned sequence numbers as they are entered, and these are used for later references.

The process of editing code online is considered by some to be the heart of a good online system. Thus, various schemes for modifying programs have been developed. These vary from those which use concise control languages on a keyboard to insert, change and delete lines by line number [6], to those which use more advanced methods utilizing displays and editing by context rather than line number [7, 8]. In any case, the ability to rapidly modify a program at a console is an extremely important one for online users.

Another consideration in the preparation of programs in an online system is the one concerning the preservation of and access to files entered at a console. Since the console itself preserves little more than a written list of the transactions that have taken place, a mechanism is needed, to provide later access to disk or other mass storage device files made in this way. Among some of the other requirements are those for assignment, protection from destruction, privacy and purging to permit new files.

Compiling-interpreting. Assuming one has a capability for entering and modifying code, the next requirement is that of compiling or interpreting so that the code may be executed. One of the first observations that can be made about online compiling is that one needs a rapid compiler. Since one can, in the normal course of an online sitting, compile several or more times, and since it is extremely unsatisfying to sit and stare at a quiet console for long periods while a long run is taking place, the need for speed is readily apparent. This aspect is important enough for the user to forego some of the advantages that might be achieved with a slower compiler, such as extremely efficient code and complete listings. This requirement is compounded by the fact that most online systems are time-shared, causing lengthy compute times to be multiplied considerably in elapsed time, resulting in intolerable delays for very long jobs. Techniques for speeding up compilers include limiting them to "one pass," linking at load rather than compile time, "incremental" compilers—where single statements can be compiled and added to an existing program—and the maintenance of one compiler for program checkout and another when the program reaches the production stage.

Another aspect of online compiling that has been found to be valuable is that of interaction between user and the compiler. With this, the compiler can query the user regarding what it considers to be error conditions, permitting the user to change the program before the compilation is complete. This interaction could be extended to include questions that would aid the compiler to produce better code.

The use of interpreters online has proven to be of some value. Interpreters can find many errors during execute time that are difficult or impossible to detect immediately with a compiled program. Also, interpreters can, in general, be more helpful to users, and for various reasons, interpreters are easier to produce and modify then compilers permitting easier language experimentation. Of course, interpreted programs are usually slower than compiled programs, as much as 40 times slower in some cases that have been examined. Also, the interpreter occupies space that the program or its data could otherwise occupy.

Executing-debugging. The ability to detect and correct an error while working online is one of the most exciting aspects of this technique of computer utilization. This rapid exchange must be aided by a reasonably good online debugging system. (Examples of these are found in [9, 10]). Characteristically, these systems permit the inspection and modification of program and data simply and concisely. They also permit modification of program paths and searching for specified values so that particularly difficult situations can be investigated.

Of course, the characteristic use of online debugging does not admit lengthy printouts that are commonly used in offline debugging. There are cases, however, where large dynamic printouts are valuable. Taking these dumps on tape and having them printed later is no great problem for users located near the computing facility, but this procedure is somewhat less valuable to those located at a considerable distance.

With an online language which permits easily added console input-output statements, a certain amount of debugging information can be attained from a "conversation" with the program while it executes. In fact, a good deal of console debugging consists solely of observing the progress of the executing program through its output to the user.

Summary

The most interesting aspect of the online programming process is the tremendous compression of time that is possible. The entire process from coding to checkout can be repeated several or many times within a few hours. This is a process that can take days or weeks in more conventional, offline systems. Without the presence of a number of supporting tools and techniques, however, the mere existence of online consoles would not assist the user very much. In providing a system such as this, one must consider both the methods of operation forced upon the user by his environment and those that should be present to take advantage of the situation. When this is done, a user can accomplish more in the period of time when he is present than he could by the indirect approach presented by offline computing techniques.

REFERENCES

1. Schwartz, J. I. Programming languages for on-line computing. Proc. IFIP Congress 65, New York, 1965.
2. Shaw, J. C. JOSS: A designer's view of an experimental on-line computing system. Proc. Fall Joint Comput. Conf. 1964, Vol. 26, Pt. I, pp. 455–464.
3. Culler, G. J., and Fried, B. D. An on-line computing center for scientific problems. M19-3U3, TRW Computer Div., Thompson Ramo Wooldridge, Los Angeles.
4. The MIT Computing Center. *The Compatible Time-Sharing System, A Programmers Guide.* The MIT Press, Cambridge, 1963.
5. Schwartz, J. I. The SDC time-sharing system, part I. *Datamation 10,* 11 (1964), 28–31; Pt. II, *10,* 12 (1964), 54–55. (Also available as SDC doc. SP-1722, System Development Corp., Santa Monica, Calif.)
6. Aranda, S. M. EDIT. SDC doc. TM-1354/444/00, System Development Corp., Santa Monica, Calif., Feb. 16, 1965.
7. Daley, R. C. ED, a context editor for card image files. CC-245, MAC-M-195, Comput. Ctr., MIT, Cambridge, Nov. 20, 1964.
8. Saltzer, J. H. TYPSET and RUNOFF, memorandum editor and type-out commands. CC-244-2 MAC-M-193-2, MIT Project MAC, Cambridge, Jan. 11, 1965.
9. EDP-6 time-sharing software. I-61B, Digital Equipment Corp., Maynard, Mass., 1965.
10. Command research laboratory user's guide. SDC doc. TM-1354 series, System Development Corp., Santa Monica, Calif.
11. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Comm. ACM 9* (Mar. 1966), 143–155.

DISCUSSION

Sammet: There is a big difference between using a system online with the enormous convenience that you get and the concept of just about not being able to solve the problem otherwise. Where the situation is one in which you are trying to generate the mathematical expressions, in many cases, you either can tell ahead of time—or don't really care—what the form of the expression is; that's what you want the computer to do for you and a batch situation can be defined. But there are other cases in which you can't tell what the result of an operation, say differentiation, is going to be until you physically see it. Here is the time in which you need the online situation, in which not only is the initial language essential, but you must have the online situation.

Furthermore, I do not agree with your point that all the languages tend to be in the batch version. There are cases and there

are things we know of, where there are additional language facilities needed for the online situation which are meaningless in the batch environment. I think the answer is that all of our experience has been with FORTRAN and JOVIAL. As we get into new areas, I believe we will need new commands.

Schwartz: If a mathematician is going to use a language in his everyday work, he would much prefer to have a language online that is certainly a different kind of language than we generally program in. It is certainly not our intent that the users be programmers. We are, as time goes on, developing tools amenable to nonprogrammer usage. We have, for example, the police department detectives, who don't know anything about computers or programming, using it for applications.

Schneider: First a word in defense of interpreters. While the interpret package itself will take up space, the actual code itself being interpreted is much more compact. Also the QUICKTRAN system, which is an interpreter, does provide traces, printouts of a variable every time it changes, and so forth.

Schwartz: TINT does that too. It is an interpretive system. It is a particular version of JOVIAL. The kind of dump I am talking about you may get over a teletype, but it may take a couple of hours to a few seconds or minutes of computing.

Book: I would like to take exception to one point you made. With online programming we tend to write and to be able to check out the largest program possible all at once. Debugging facilities, and facilities to edit and compile fast and be online, give you such a hold on being able to comprehend problems that come up in debugging that you automatically can write the program all at once.

Schwartz: I think that you may be right. Our weakness is the ability to gather routines. That might also influence what you are saying.

Bachman: Who uses time-sharing systems? We have the good fortune of having one in Phoenix that we can use. And my experience has been that everybody and his brother has been using it—engineering people, the finance office; they all use it. In fact it's usually a problem to walk around and find an open terminal; in my own setup, about 10 people out of the group of 40 were using it for one purpose or the other.

The other side that I don't think has been emphasized sufficiently is the reactive aspect of this online computing. I have been doing a lot of simulation work and I started, really not knowing when it was going to stop; so by observing the results I could tell when it had gone far enough. With an offline system it's hard to tell when to stop unless you do additional programming to set up limits for calculations. One thing that we don't have today in our present system is the ability to say stop and go somewhere else.

Opler: As part of your research department, have you made any careful plans or studied the development of equivalent programs in batch groups and in time-sharing?

Schwartz: We have just started a test, an experiment, comparing online vs. offline debugging. As always in these experiments there are many difficulties. For example, in our case there is no good offline system in the computer we have. We've had to manufacture what we think is a good offline system with two-hour response time guaranteed, and a number of other things like this. We have just begun a pilot study to see what problems we are going to have; the only reaction so far is that the user doing the offline at the moment is complaining about the two-hour response time.

Bachman: Just one point that hasn't been discussed yet, which turns out to be a very ticklish one: that is, accounting for a system like this. There is the problem of file space, which people use and sometimes use badly. Also, there is the problem of edit time, which is one function of the computer, and also problems of computing time. Now one thing that we have not found a very good answer for yet is how to account for these or how to keep track of them.

What really happens is that the ultimate user walks in with a problem that is a pressing one; he has some ideas but has not developed the concepts and selected the numbers which are going to allow you to do something for him. He starts out with an airline reservations system and as soon as he discovers the great amount of raw material within the airline reservation system which could help him with management planning of something else, or for which there is interaction between two areas of his business, then he certainly would like to have additional features. Now some of the initial airline reservation systems were built specifically for airline reservations and didn't use general purpose computers. It was rather embarrassing when they couldn't even get the data that was obviously within the system.

This evolution, then, I think, is one of the features of real-time systems, or many of the real-time systems: the burgeoning out of the real-time to involve things which are not necessarily real time. You rapidly get to develop the hybrid system, something which has a bulk data processing job which slowly moves past the facility but in the foreground there are sorts of real-time or intermediate-time aspects in support of people who want immediate reports out of the system. Frequently one of the big problems is the ability to handle this interaction between the bulk data processing job going on in the background and the jobs going on in the foreground. I don't think we yet have the languages which will address these problems.

The kind of languages which I see that we need, then, are first, data transformation languages. But second, there are the languages of resource control. These are associated more with the logistics of getting a problem done than with the logic of getting data transformation done. These languages turn out to be not available using the normal compilers but are available, in most of the systems we work with, at the operating level. What you want done at this level has some implications for, and needs information out of the logic areas in terms of desired transformations. Most of the compilers we are familiar with do not make available to the operating system the kind of information needed to assist in solving this problem.

There is a third type of language I see coming in now. Most of the online real-time systems of the past have had a very simple syntactical vocabulary—you push buttons or you make very clean parameter inserts into what we might think of as macros. We're rapidly getting to the point where the kind of language which the user wants and the kind of activities he wants the machine to do for him cannot be provided by parameter insertion or button pushing. We're beginning to build online real-time systems with compilers or language translators which may indeed drive interpreters, but they do turn out to accept languages. In this case they are application languages. We do have need for compilers inside some of the real-time systems. It's the control and data collection types of languages which are not available, and also the I/O areas are not available adequately. Therefore, we start building our own.

Seventy-eight percent of the AF expenditure is in what you might call management-administrative support area or what some people call data processing. These areas which have been built in pieces are rapidly getting to the point where they cannot be run without becoming integrated. These systems are also moving toward having aspects of online real-time processing. To get at these problems we have to start fixing it so that the programs can indeed understand and communicate to the kind of environment they are going to be running in. The languages for the logic of how you want the data transformed is the very simplest part. But for the part which does all the logistic flow for me, controls and decides what should be done next, I just don't have enough good capability in current languages.