data-division as a single section of his program but some similar grouping of declaration statements can be adopted when using the latter language. A set of such listings together with a general flow diagram for the program suite theoretically meets in full the requirements under category 1. If 'correct' cards are used only when program alterations have been fully checked out, then the set of automatically dated listings including the card listings enables the reconstruction of a previous position.

This is, of course, a theoretical situation but it is interesting to note that all other documentation comes into category 2 and its production can be the subject of management investigation and appraisal. Some of this can be produced by automatic means—if the manufacturer does not supply an analyser for use with your operating system he should be encouraged to do so, and if you really like to see the procedural parts of a program with lines drawn round them, Grace Hopper has a solution. But beware using programmers' flowcharts for anything but lighting bonfires, they will be both out-of-date and wrong; nor is it worthwhile expending effort to try to make them otherwise as many of the typical error conditions arising during production running would not be apparent from a study of such documents, in any event. It is better to encourage an intelligent use of procedure names, comment cards and comment fields in statement cards—this latter particularly for identification of amended or added statements during updating.

Rather than insisting on more elaborate, rigorous and time-consuming documentation it is, once again, possible to determine root causes of failures in running production work. An exercise carried out over the last two years in a large commercial installation has revealed a distinct pattern in reasons for abortive work and has enabled documentation and control diagnostics to be concentrated in those areas providing the best returns. Every installation manager will express an opinion as to why abortive results occur, but a systematic recording and correlation of the events leading to each failure is a management exercise which must be gone through before any spectacular advance can be made in improving running efficiency of work. No two installations are precisely the same and for this reason blanket edicts on standard procedures which are not based on a thorough analysis of failures will only increase the operating overheads without achieving the desired result.

Acknowledgements

The author is grateful to the Chairman of the South Eastern Region of the British Gas Corporation for permission to publish this paper and to many of the Directors for providing the incentive to carry out this work.

Discussion

Job control languages and job control programs

D. W. Barron

Department of Mathematics, University of Southampton, Southampton, SO9 5NH

Introduction

It is reported (Commoner, 1971) that one of the first visible signs of the massive pollution of Lake Erie was the appearance of a thick carpet of slimy green overgrowth of algae. This will be a familiar phenomenon to anyone who has tried to run a program on a 'third generation' computing system and has found himself confronted with a thick overgrowth of Job Control Language. (Whether this can be described as green and slimy is a matter between the reader and his operating system.)

It comes as a shock to the innocent user to find that in addition to a programming language, he has to learn a *job control* or *job description* language before he can get a job through the system. Once he graduates from the simplest form of job, he is likely to find the preparation of the job control cards at least as troublesome as the program itself, if not more so. This difficulty will increase as he tries to avail himself of the more advanced facilities offered by the system and sooner or later he will give up. It seems likely that the full exploitation of the facilities of a modern sophisticated operating system is denied to all but a few by the almost impenetrable obscurity of the job control system. As Cheetham and Wickham (1973) have observed,

> 'In order to understand how to use a powerful, flexible operating system, even to run small, simple jobs, one has to be a powerful, flexible programmer'.

When the manual for a typical modern operating system (ICL, 1973) is one and a half inches thick (520 pages), it is not surprising that most users, having found a set of JCL cards that work, will go to considerable pains to avoid the necessity of change. It is interesting to speculate why user-dissatisfaction has not led manufacturers to improve their offering in this respect (or, indeed, in many others). It is another example of the passive acceptance of what the manufacturer offers: to put it crudely the users don't realise that something better is possible. (In a recent paper (Tanenbaum and Benson, 1973) reference was made to a system in which '. . . . the standard FORTRAN compile, execute, debug and return to the editor sequence is 38 cards.' One can only assume that the people who designed such a system had never actually used a computer to solve real problems, and therefore considered the user interface as irrelevant.)

There are, of course, some *aficionados* for whom JCL brings back the esoteric mystique that high-level languages have largely removed from programming, but we do not consider them as relevant to the subsequent discussion.

JCL as a language

Although the initials JCL stand for job control *language*, it is rarely considered in the same way as a programming language. The JCL statements are not clearly distinguished from the facilities they control, and this is the cause of much of the trouble. It is a cardinal principle of language design that you should decide what you want to say before you think about how you are going to say it. In the case of IBM's JCL it is all too clear that they did it the other way round, and the language format is entirely determined by the decoding mechanism, which obviously borrows much from a macro assembler.

```
//COMP     EXEC PGM=IEYFORT,PARM='SOURCE'
//SYSPRINT DD   SYSOUT=A
//SYSLIN   DD   DSNAME=SYSL.UT4,DISP=OLD,
                DCB=(RECFM=FB,LRECL=80,
                BLKSIZE=800)
//GO       EXEC PGM=FORTLINK,COND=(4,LT,C)
```

Fig 1. Extract from OS/360 JCL for a simple FORTRAN job.

Examples abound: two will suffice. **Fig. 1** shows part of the JCL for a simple FORTRAN job. The EXEC statement includes the phrase

COND (4, LT, C)

which is interpreted to mean 'omit this job step if 4 is less than the condition code returned by Step C'. It requires distinct effort to work out what condition codes cause the job step to be included. In the first place, the comparison is backwards way round: the natural way is to write the variable first and the constant second, thus 'C greater than 4'. Also, it would be more in the spirit of programming languages to specify the condition for *inclusion* rather than *exclusion*, or at least to replace the word COND by OMIT IF.

The second example also from Fig. 1 comes from the data definition statements (e.g. SYSLIN), where the syntax consists entirely of brackets and equals signs. The best that can be said of this is that it is an implementation in search of a language.

(It may be argued that these are small irritations, but the sum of a large number of small irritations is sheer unacceptability, unless one accepts the converse argument, put forward by Stafford Beer, that one last desperate approach to the problem of saving a crumbling church tower is to ask the death-watch beetles to hold hands.) The next example is taken from the CDC 7600 (SCOPE 2) system, in which one may find oneself obliged to write a job control statement of the form LGO , , , , LONDOP. The four commas indicate that 'LONDOP' must replace the fourth parameter in the PROGRAM statement. As a language, this is on a par with the grunts by which Neanderthal Man communicated with his fellows: the chaos that can be caused by omitting a comma is too awful to contemplate, being rivalled only by the remarkable effects that can be caused by a single spurious blank in the middle of an OS JCL statement.

GEORGE 3 command language (ICL, 1973) does represent a significant step forward in that it uses concepts well-known in programming languages, the conditional, the GOTO and the procedure. Thus, whilst OS JCL only allows the conditional inclusion (or is it exclusion?) of a single job step, GEORGE 3 has labels, GOTO's and IF statements, which greatly increase the power of the language.

In earlier versions of GEORGE 3, the IF qualified only one command (as in ANSI FORTRAN): this meant that the only conditional used in practice was a conditional jump. (More elaborate constructions could be achieved by making the IF control the call of a macro, but this was a roundabout way of getting things done.) The most recent version of GEORGE 3 allows IF . . . THEN . . . ELSE as well as IF . . . THEN . . . However, it is symptomatic of the way in which designers of job control languages have ignored the lessons of high-level language design that the designers of GEORGE 3 have fallen into the trap of the 'dangling else' that has been known for ten years. Thus the GEORGE 3 user can write

IF . . . THEN . . . IF . . . THEN . . . ELSE

which can be interpreted as

IF . . . THEN (IF . . . THEN . . . ELSE . . .)

or as

IF . . . THEN (IF . . . THEN) . . . ELSE . . .

The main defect of the GEORGE 3 job description language is its syntax which, as can be seen from **Fig. 2** is rather minimal. The general pattern is that each command starts with a verb, which is followed by parameters separated by commas. However, as shown in Fig. 2 unexpected commas are sometimes required, which is a trap for the unwary.

Hidden underneath this simple syntax are many of the attributes of a high level language.

Thus, whilst OS JCL has 'catalog procedures' which are precisely analagous to assembler macros, GEORGE 3 has things called macros that are in fact true procedures. (The distinction is that a true procedure has a local name-space and consequently procedures can be nested with no fear of name clashes.) It is ironic that both IBM and ICL should have chosen precisely the wrong word to describe this facility.

If we accept that JCL is a language akin to a programming language, then we see that in principle it can be interpreted or compiled. Interpretation is the more usual, but it can lead to situations where, on a system dominated by small jobs, up to 30% of the processing capacity can be expended on JCL interpretation.

The relation between job control and programming

In an earlier paper (Barron and Jackson, 1972) the evolution of job-control languages was traced, and it was suggested that OS/360 JCL is akin to an assembly language with macro facilities, whilst GEORGE 3 Command Language is a rudimentary high level language. A possible line of development is to pursue this analogy and see what in job control corresponds to a really high level language. It is sometimes claimed that this amounts to inventing programming all over again, but this is not true: what we are saying is that job control is just another sort of programming, and a job control language should be viewed as a programming language for a processor with an unusual repertoire of basic operations. The question that arises is thus whether this new form of programming can be accommodated within existing languages, or whether an entirely separate and different language is needed for the purpose. If we conclude that a separate language is required, then we can view the operating system as the implementation of that language. (I am indebted to Mr. David Beech of the IBM Laboratories at Hursley for this illuminating observation.)

Do we need a job control language?

As an alternative to pursuing the consideration of what might be included in a high-level job control language, we can consider a more radical alternative: can we do away with

```
      LOAD MYPROGRAM
      ASSIGN *CRO,MYDATA
      ASSIGN *LPO,MYRESULTS
      MONITOR ON, DELETE
      ENTER 0
      IF NOT HALTED (LINE PRINTER), GOTO 999
      ONLINE *LP1,GREMLINS
      RESUME
999   IF FAILED, LISTFILE MONITOR, *LP1
      IF HALTED, ENTER 1
      ENDJOB
```

Fig. 2 Example of GEORGE 3 Job Description. (The program in the file MYPROGRAM is to be run using data from the file MYDATA and sending results to file MYRESULTS. If the program halts with the message 'LINEPRINTER' a line-printer is to be put on line and the job resumed. If the program halts for any other reason it is to be restarted at entry point 1; and if it fails the monitor file is to be printed. Note the apparently inconsistent appearance of commas.)

JCL? This is certainly possible in single-language systems (e.g. OS-6 (Stoy and Strachey, 1972) POP-2 (Burstall *et al.*, 1971), APL (Pakin, 1972), BASIC (Kemeny and Kurtz, 1967). In the more general context a more amenable approach is to design systems in which we can implant subsystems that require a minimum of JCL: this opens up the possibility of absorbing the residual JCL into the various programming languages. We may note that in existing systems some job control information has already been absorbed in a fairly haphazard way by *ad hoc* extensions to existing languages. Thus in FORTRAN, I/O channels are identified by number, e.g.

WRITE (6, 100) X, Y, Z .

The 'unit number' must somehow be associated with a specific device or dataset: in the ICL 1900 version of FORTRAN this is done in the 'program description segment' by a statement of the form

OUTPUT 6 = LP0 .

(The program description is a form of bastard JCL, read by the compiler.) However, if when running under GEORGE 3 the user wants his output to go to a file, he must provide a piece of GEORGE 3 job control also, thus:

ASSIGN * LP0 TO : MYPRINTFILE .

This arbitrary division of function is confusing to the programmer, who may be pardoned for thinking that he exists for the convenience of the system, rather than vice-versa. He would be much happier with a piece of genuine FORTRAN, thus:

CALL OPENFILE (6, 11HMYPRINTFILE) .

(This latter type of call was provided in the GEIS conversational FORTRAN system.)

Having once arranged things this way, many incidental benefits appear. The second parameter does not need to be an explicit text constant: it can be a variable or a function having a text constant as its value. It can even be a variable whose value is set by a preceding READ statement, thus preserving the flexibility of file-naming but in a context more familiar to the programmer.

Minimum JCL versus high level JCL

Many of the troubles that beset large scale operating systems can be ascribed to the fact that they try to be all things to all men (Barron, 1972). Similarly, the complication of a job control language stems largely from the fact that they cater for the most complicated kind of job, and lose sight of the indispensible maxim that simple things should remain simple. (The obsession with generality is not new: it was pointed out long ago by Needham (1964) that the classic von Neumann machine is based on the assumption that users will wish to write random programs. It is only recently that the inherent structuring of programs has been reflected in machine design.)

In fact most user communities will generate jobs that largely conform to a pattern. For example, consider the following extract from the Leeds University 1906A User's manual (Hock, 1971):

> 'PROG is a general purpose macro designed to enable the user to translate and run his programs without having to use GEORGE job control language. With this macro, a user programming in any of the languages in accepted use at Leeds can translate and run his program . . .'

Here a macro has been used to adapt the system to the class of users, but the implication is that the facilities provided are more general than is required, and a more efficient means of adapting would be preferable.

We thus see that the problem really moves back into the design of the underlying operating system. To quote Cheetham and Wickham (1973) again.

> 'A system designed to cope with large complex jobs takes so much of the resources of the machine to discover that a job is small and simple that no job ever is (small and simple).'

This is why the concept of subsystems is important. If the majority of jobs are FORTRAN compile-and-go, then a system of the WATFOR type (Shantz, 1967) will not only run more efficiently, but will also relieve the user of the need to provide elaborate job control statements. Even if an in-core compiler is not available, unless he actually wants to preserve intermediate material the user should not have to include (to him) irrelevant job control information about link editing, nor is it desirable that he should have to retrieve a macro (catalog procedure) to get round this.

Linear jobs

If we look at the functions of JCL, we can see that they fall broadly into two groups:

1. sequencing a number of job steps,
2. establishing the environment for a job step (i.e. file opening and creation, etc.).

A large number of jobs consist of a number of job steps that are to be executed in sequence, with the option of stopping after any step. We call such a job a *linear job*. The prime example is compile-link edit-go. Such jobs are suitable candidates for a minimum-JCL system: if the sequencing is of a more complex nature then we need the constructs of a high-level language (e.g. **if-then-else, repeat while, case**) to control the sequencing in a palatable manner.

Let us consider the classic compile-link edit-go job as the paradigm for this class. At the end of the compiling phase, if there have been no errors the linkage editor must be loaded and entered to link together the compiled segments and any library segments that have been referenced. In OS/360 this is achieved by a JCL card which includes a reference to a condition code returned by the compiler (see Fig. 1): there are also JCL cards specifying the files (datasets) to be used as input to the linkage editor. But the compiler knows whether there have been any errors (and hence whether or not to call the linkage editor), and it (presumably) knows where it put its output (which will be the input to the linkage editor), and thus it is not strictly necessary to exit to JCL-decoding level. We can dispense with JCL entirely in this context if it is possible for the compiler to delete itself and to load the linkage editor, passing information across in the machine's registers (or in a reserved area of store).

This principle can be extended to most linear jobs. It is provided in a restricted form in ICL 1900 series compilers (under the name 'automatic consolidation'). The PDP/10 Monitor uses a similar technique to provide a powerful compiling subsystem in which, for example, compiling of a source file is skipped if there exists a file containing a relocatable binary version of the same routine, at least as recent as the source.

The necessary feature in a system to permit the construction of JCL-less subsystems is thus the ability of a program to delete itself and nominate a successor, retaining intact its environment (i.e. open files, register contents). This implies that an effective way exists of passing information from one program to another; a method of creating and using temporary files suffices. It helps if there is a uniform way of handling character files.

The linear job facility is found at its most elaborate in the Titan operating system (Hartley, 1972), where it is known as 'changing phase'. However, since the facility was added subsequent to the original design, the mechanisms for communicating between job steps are less elegant than one would desire.

Before leaving this topic, we may note as an aside that the

linkage editor usually has the same obsession with generality that we have noticed elsewhere, and assumes that every segment of a program will have been written in a different high level language. Dramatic improvements can be made simply by treating the most common situation as a special case (Barron, 1976). In a FORTRAN-oriented establishment the continued survival of the linkage editor is a consequence of the sacred overtones attached to the concept of independent compiling of routines. In fact, this concept was only significant when compilers were slow and cumbersome. Nowadays an in-core FORTRAN system will compile source code faster than a linkage editor can process the equivalent pre-compiled form, and provided that there exists a convenient file-substitution system (Poole and Lang, 1968) to facilitate the inclusion of library material in source form, the persistence of linkage editing in large systems is to a large extent a triumph of faith over reason.

High level JCL

For situations more complicated than the linear job, we need a high-level JCL. We examine here the facilities that the operating system should provide to make this possible.

The user interface in the operating system

Consider the functions of an operating system. There is a great deal of behind-the-scenes activity in resource allocation, scheduling, interrupt processing, etcetera. But from the point of view of the individual user, the system allows (or should allow) him the following facilities:

(*a*) to control the sequencing of a number of related operations;

(*b*) to call programs (compilers and utilities) into action, providing values for various items of parametric information;

(*c*) to provide an environment for these programs by relating symbolic names to actual files or devices;

(*d*) to set up and manipulate files, directories, etc.

These are closely parallel to the things we want to do in programming, but at an early stage in the development of operating systems, designers set off along a lengthy blind alley though not appreciating this fact.

The Atlas job description (Haworth, 1961) was regarded as a static definition of an environment in which a piece of program was to be run. The same is true of the dataset description cards in OS/360 JCL. However, most of the setting up can in principle be done by the program itself. Some of it will be initialisation, but some can be deferred until a later stage, and may be conditional on the outcome of earlier processing. For example, if a program wishes to output to a printer there is no need to specify this before the job is started—it can be done at run time by a suitable extracode (supervisor call). Having taken this step, it is apparent that the static job description is merely a formalised way of describing the initialisation of certain parameters, and that the 'job-description decoder' is merely an interpreter for a stylised language, all the work being done by the same supervisor calls that the program could use at run-time.

With the benefit of hindsight, it seems obvious (to the author at least) that the basic interface between the user and the system should be defined by a number of supervisor calls that provide the basic job control language, exactly as the operations wired into the processor define the machine language. The choice of the correct set of primitives is a major design exercise, as is the definition of a systematic set of conventions for the way in which the primitive functions return information about their success or failure.

The process of job control now reduces to the generation of an appropriate sequence of these supervisor calls. An implication of this is that anything that can be done by job-control language can also be done at run-time by supervisor calls. It is illuminating to see how this dynamic facility is viewed in various systems. In OS/360 many, but not all, of the things that can be done by JCL can also be done by obeying supervisor calls (generated by assembler macros). In GEORGE 3 it is possible to do almost anything at run time, but to do it the program must generate a string of job-control statements in character form that will be treated exactly as if they had just come from an external peripheral. Thus the program must go to the trouble of encoding information which will immediately be decoded again by the job-description interpreted. This is not just an inefficiency: it implies that the job-description decoder has been elevated to an unnecessarily central and indispensable position. Perhaps the system that came nearest to the ideal proposed was CTSS (Corbato, 1963): once one had logged-in and entered a program, anything that could be done from the keyboard could be done by supervisor calls.

We thus have the concept of an environment being set up dynamically. Some aspects can be dealt with as required, but others (e.g. time and space limits) must be set up initially, and must be subject to some protection (or at least must interact with an accounting and budgeting system). Thus we are led to associate with every program an *initialisation phase* (the idea comes from the CTL E4 executive (CTL, 1973)) which need not, of course, be in the same language as the rest of the program.

In fact the interface presented to the user by the operating system must also include a mechanism for communication between job-steps. There must be associated with each job a communication area, with a mechanism to permit job steps to insert and extract messages. The PDP-10 monitor system provides a reserved area of store for such communication; alternatively supervisor calls (extracodes) can be provided to read and write a set of pseudo-registers. In either case a suitable compiler can map this communication area into a set of JCL variables. An alternative approach is to achieve inter-job step communication by one or more pseudo I/O channels along which job steps can exchange messages. The 'event' mechanism of GEORGE 3 is an example of this approach. As described in an earlier paper (Barron and Jackson, 1972), a program running under GEORGE 3 can (voluntarily or involuntarily) cause an *event* to occur, and associate a character string with it. The occurrence of an event can be tested by a JCL conditional statement, e.g.

```
IF HALTED ....
```

The associated message may be used as part of the conditional test, e.g.

```
IF HALTED 'ERROR' ...
```

will test for the event 'HALTED' and a message starting 'ERROR'. Within a procedure (macro) the message can be used in a more elaborate way to provide a genuine transmission of information between job steps. For example, suppose we have a program that is going to decide its successor on some dynamic basis. We arrange for it to generate the 'HALTED' event with a message 'LOADuvwy' where uvwy is the 4-character name of the successor. Then within a procedure (macro) written to control the original program we can write:

```
        IF HALTED 'LOAD' GOTO 9991
        ....
        ....
9991    SETPARAM A, MESSAGE(5, 8)
        LOAD %A
        ....
        ....
```

Within a macro, SETPARAM α, β replaces the value of parameter α by β, so in the above example parameter A is replaced by characters 5-8 of the message, then LOAD %A loads the

appropriate successor, % being the marker for a formal parameter.

It will be noted that GEORGE 3 events are 'polled' by the job description. A very powerful and attractive feature of an operating system interface would be the ability to generate events as interrupts, so that in the job control we could write statements like PL/I ON-conditions, e.g.

ON HALTED 'ERROR' GOTO 9999

to set traps for certain events. (GEORGE 3 provides a WHENEVER command, but the only condition that can be tested this way is COMMAND ERROR.)

Job control programs

If the operating system is structured in this way, selected aspects of job-control can readily be absorbed into an existing language system to give a neat user interface. Moreover, almost any language can be used as a job control language provided that it can be compiled into, or interpreted in terms of, the appropriate set of supervisor calls. It would thus appear that we have removed the need for a specific and unique job control language, and opened the way to machine-independent job control.

This is true, but we must beware of falling into the trap of thinking that because the need for a job control *language* has been removed, the need for a job control *program* has also vanished. In an online system, whenever an action has been completed control returns to the key-board for the user to specify what to do next. Similarly, in a 'batch' environment, at the end of a job step control must return to a job control program which decides what to do next (unless the job is a linear job as defined above). The JCL of OS/360 or the job description of GEORGE 3 are essentially job control programs written in a special-purpose language. What we have shown is that the operating system could be designed in such a way as to allow job control programs to be written in almost any language; this might or might not be the same as the language used in the actual job steps.

Short term improvements

We have indicated a system structure that would greatly facilitate the provision of an attractive method of job control. But this depends on changes in operating systems, which is a long term prospect. In the long term, as Lord Keynes liked to remark, we shall all be dead. For the short term an attractive approach is to apply syntactic sugar to an existing job control system. That is to say, a palatable job control language is designed together with a program to translate it into the native JCL of the machine. One such system has been described by Newman (1972) for the GEORGE 3 operating system on the ICL 1906A. It is suggested that the system is portable and applicable to other operating systems but no examples are given apart from 1906A use. Dakin (1973) has suggested a machine independent job control language that can be translated by an intelligent satellite terminal into the job control language any one of a number of remote mainframe operating systems, and Parsons (1974) has independently designed a similar system.

Acknowledgements

The ideas presented in this paper have evolved over some two years. During this period I have tried out versions of the ideas on audiences up and down the country, and I am grateful to all those who, by participating in the subsequent discussions, assisted me to clarify my thoughts.

Postscript—MU 5

After this paper was completed a presentation of some features of the MU5 system by Morris (1974) made it clear that the MU5 job control system is constructed along the lines advocated in this paper. With hindsight it is possible to see that this is implied in the published description of the MU5 operating system (Morris *et al.*, 1972), though the point is not brought out explicitly. Once again the Manchester team have introduced a revolutionary concept with such a degree of understatement as to go unnoticed.

References

BARRON, D. W. (1972a). What Happened to Operating Systems. *Proceedings of the Software 72 Conference*, pp. 18-21. London: Transcripta Books.

BARRON, D. W. (1972b). *Assemblers and Loaders* (Second Edition). London: Macdonald/New York: American Elsevier Inc.

BARRON, D. W., and JACKSON, I. R. (1972). The Evolution of Job Control Languages. *Software- Practice and Experience*, Vol. 2, No. 2, pp. 143-164.

BURSTALL, R. M., COLLINS, J. S., and POPPLESTONE, R. J. (1971). *Programming in POP-2*, Edinburgh: Edinburgh University Press.

CHEETHAM, C. J., and WICKHAM, R. (1973). Cafeteria Systems. *IUCC Newsletter*, Vol. 2, No. 1, pp. 6-11.

CTL. *E4 Product Description*, Document Reference 381/47, Computer Technology Ltd.

COMMONER, B. (1971). *The Closing Circle*, London: Johnathan Cape.

CORBATO, F. J. *et al.* (1963). *The Compatible Time Sharing System: a Programmer's Guide*, Cambridge Mass: MIT Press.

DAKIN, R. (1973). Private Communication.

HARTLEY, D. F. (1972). Techniques in the Titan Supervisor. In *Operating System Techniques*, ed. C. A. F. Hoare and R. H. Perrott. London: Academic Press.

HOWARTH, D. J. (1961). The Manchester University ATLAS operating System, Part II. *The Computer Journal*, Vol. 4, pp. 226-229.

HOCK, A. A. (Ed.) (1971). 1906*A User's Manual*, University of Leeds Electronic Computing Laboratory.

ICL (1973). *Operating Systems George 3 and 4*, Technical Publication No. 4345. London: International Computers Ltd.

KEMENY, J. G., and KURTZ, T. E. (1967). *Basic Programming*, New York: John Wiley and Sons Inc.

MORRIS, D. (1974). Presentation at the British Computer Society Symposium *Job Control Languages, Past, Present and Future.*

MORRIS, D. *et al.* (1972). The Structure of the MU5 Operating System. *The Computer Journal*, Vol. 15, pp. 113-116.

NEEDHAM, R. M. (1964). The Exploitation of Redundancy in Programs, Conference on *The Impact of Users' Needs on the Design of Data Processing Systems.* Proceedings, pp. 6-7. London: Institute of Electrical Engineers.

NEWMAN, I. A. (1973). The Unique Command Language-Portable Job Control, *Proceedings of DATAFAIR* 73, pp. 353-357.

PAKIN, S. (1972). *APL*/360 *Reference Manual*, Palo-Alto: Science Research Associates Inc.

PARSONS, I. A High Level Job Control Language. *Software-Practice and Experience*, in press.

POOLE, P. C., and LANG, T. (1965). The Development of On-Line Computing Facilities for the KDF 9, Part I, *The Computer Journal*, Vol. 11, pp. 5-11.

SHANTZ, P. W. *et al.* (1967). WATFOR—The University of Waterloo FORTRAN IV Compiler. *CACM*, Vol. 10, No. 1, p. 41.

STOY, J. E., and STRACHEY, C. (1972). OS6—An Experimental Operating System for a Small Computer. *The Computer Journal*, Vol. 15, pp. 117-124 and 195-203.

TANENBAUM, A., and BENSON, W. H. (1973). The People's Time Sharing System. *Software-Practice and Experience*, Vol. 3, No. 2, pp. 109-120.