Design for a transportable job organisation language

L. Moore

Department of Computer Science, Birkbeck College, University of London, Malet Street, London WC1E 7HX

The design of a transportable JOB Organisation Language called JOBOL is presented; the design adheres to a set of desirable criteria which determine the scope of the language, its fundamental entities and operations, and its programming structures. The semantics are described informally but a formal definition of the syntax is given in an appendix. Illustrative examples of JOBOL programs are given.

(Received February 1977)

1. Introduction

The situation with regard to job control languages today is in many ways analogous to that of programming languages twenty years or so ago. Previously every machine had its own programming language whilst nowadays one expects to find a choice of several common high level languages available at any medium sized computing installation. Prior to the development of high level languages the concept of machine independent programs could not be realised. Now, however, we find that machine independent programs are not transportable *per se*, since we need to transport not merely a program but rather a job, of which the program forms a part (Barron and Jackson, 1972; Moore, 1975).

From the user's point of view, even a machine independent program is written for some particular virtual machine. Its *machine independence* denotes independence of machine hardware design and operating system, not independence of resource environment in which the program is to run. This resource environment is the virtual machine for which the program was written, and it must be described either within (Frank, 1976; Morris, 1974) or along with the program. Many programming languages have no facilities for providing this information within the program, hence the need for a language which will cope with this job description area of job organisation (Howarth, 1962). Moreover, a job may include more than one program. It may be broken up into steps, which are to be executed in a specified sequence, each step requiring modification of the resource environment. These steps are concerned with reorganisation of that environment, general management of files and control of step sequence (CDC, 1969; ICL, 1973; IBM, 1967). It is precisely these areas of information, with regard to the description and control of a job, which fall within the general area of job organisation.

The job organisation language, JOBOL, is designed to simplify the problems of communication between the user on the one hand (Palme, 1969), and any one of an undefined set of operating systems on the other. Therefore, the scope of JOBOL is restricted as follows:

(*a*) It is capable of being precompiled

(*b*) It avoids requiring facilities which cannot be provided except by modifications of, additions to, or rewrites of the operating system

(*c*) No consideration is given to interactive requirements as such. This is partly a corollary of (*a*).

(*d*) It is capable of reasonably simple implementation in an efficient way.

The question is often asked whether there should be a special purpose language for job organisation? To this question there have been a number of affirmative answers (Colin, 1971; Barron and Jackson, 1972; Newman, 1973; 1975; 1976). However, another seemingly diametrically opposed philosophy should also be mentioned. The central idea of this philosophy is that each of the basic operations of job control should be available as (precompiled) procedures, and that these procedures may be activated by calls from programs written in any programming language whatsoever (Barron, 1974; Frank, 1976; Morris, 1974). If this be so, it is argued, then there is no need for a separate language especially concerned with job organisation as such.

The type of solution produced by this philosophy does appear to be fundamentally different from the type of solution proposed by the author (Moore, 1975). However, this is scarcely surprising, since it is a solution to a different problem, namely the design of a new, complete computing system, not the problem of transporting jobs from one computing system to another. The apparent difference in view may therefore turn out to be more superficial than fundamental.

It is asserted herewith that, although the scope of JOBOL (or any other job organisation language) should be restricted to the area of job organisation operations, the converse is false. The scope of job organisation operations is much wider than that of job organisation programs. For example, consider the problems involved in the design and writing of a program which is concerned with the management of a large data base containing many collections of files. The program maintains an index hierarchy to facilitate read/write access to the files and it provides facilities for addition, deletion and updating of files. In choosing a language in which to write such a program, the programmer would surely look for a high level general purpose language with wide ranging facilities, including, in particular, the ability to make calls of just those procedures (of job organisation) already mentioned. Not JOBOL, nor indeed any other job organisation language would be adequate for such a program. It is not a job organisation program, and yet it is necessarily and intimately concerned with operations of job organisation.

It is therefore maintained that the philosophies of JOBOL and of the procedural approach, far from being incompatible, are highly complementary. It is consequently hoped that the JOBOL proposals contribute positively to the problem of determining the required procedures.

2. The main features of JOBOL

2.1 *Basic entities*

JOBOL is a relatively simple language whose scope has been restricted deliberately to the *sine qua non* of job organisation; this is considered to be a merit since it directs design attention towards the central issues. The basic entities of the universe of JOBOL are files and strings; however, there is a fundamental difference in the way in which these two entities are treated. A string is a vehicle merely for receiving certain pieces of non-

universal information from the user, and transmitting this information (*untouched by hand*) to the operating system, compiler, or any subsystem which requires it. The information is called *non-universal* because its form is tailored to meet the requirements of a particular system. If a JOBOL program were transported from one computing system to a different one, then all the strings which occur in the program would require careful examination and probably alteration. A simple example of such a string is the parameter list specifying compiler options when a compiler is called. The JOBOL system does not manipulate strings, it merely transmits them.

Files, on the other hand, are the one entity upon which the JOBOL system does operate. From the JOBOL point of view, a file is a logical unit of data used as an operand in a JOBOL program. It may be stored internally on disc, magnetic tape, or any other medium regarded as part of the computer memory. A file of data may be input to a program or output from a program or it may itself be a program. It is a logical unit, not a physical unit. A file has an Identifier, a Contents (i.e. an unstructured unit or chunk of data) and a Descriptor. The Descriptor is a data structure containing all the available information relating to the attributes of the file. This information is required by the operating system so that it may access, manipulate and dispose of the file.

From a mathematical/linguistic point of view, it should be said in short that there is only one class of variable in JOBOL, namely the file. In addition, there occur denotations for constants, namely strings. When a job is to be transported from one environment to another it is the constants which may require change, being the machine dependent part of the JOBOL program. These (string) constants embody, so to speak, the initial conditions.

2.2 *The user's image of the JOBOL system*

JOBOL presents a standard interface to the computer user which is independent of the hardware and the operating system, and may be thought of as a virtual computing system. This system appears to the user as a collection of files together with a File Manager. The File Manager keeps a Directory of files. Existence (or non-existence) of a file is, by definition, equivalent to the existence (or non-existence) of an entry for it in the Directory. Each file entry in the Directory contains all the information relevant to that file other than its actual contents, but it includes, of course, a pointer or reference to such contents. The aggregate of information included in a Directory entry for a file is called the File Descriptor for that file.

The JOBOL system regards the contents of a file as the concern strictly of the user. There are not many things a user can do with a file contents. He can arrange for it to be executed (run). This assumes that its contents are executable; another way of saying this is that the contents provide a suitable data stream for another file which is executing, namely the loader.

Any executing program may require two or more files as input datastreams, and may create two or more files as output datastreams. If it does it will contain denotations within it of the datastreams required. The user will have to state, within his JOBOL program, which JOBOL file is to be used for each of the datastream denotations given within the executing program. Such information is given as part of the JOBOL command, RUN.

Another user activity with files involves the logical copying of the contents from one file to another, or from several files (sequentially) to another (i.e. concatenation). This is known as *Contents Assignment*.

The user has no direct access to the JOBOL File Director, but he may, within his JOBOL program, write a *Descriptor Assignment* which results in the File Manager making an appropriate entry in the File Directory. For the convenience of the user, the File Directory contains permanent entries for a number of standard file descriptions, e.g. one for unit records, one for nine-track magnetic tape, one for a file destined for the lineprinter, etc. The user may use any of the appropriate standard file names to obtain suitable file characteristics by *default*. Alternatively, he may write a Descriptor Assignment in his JOBOL program, which assigns to a file name of his choice precisely those attributes which he requires.

The expression on the right hand side of a Descriptor Assignment may be the name of an existing file; in such a case the new file acquires the same attributes. Alternatively, the name on the right hand side may be followed by a list of those attributes which (exceptionally) are different; or it may be that the name is omitted and simply replaced by a list of attributes which are required for the new file. In addition to the temporary Directory entries made by the File Manager, the user may request the File Manager to SAVE entries made during the running of a particular job, thereby creating one or more permanent files. All files created during the running of a particular job, which are not saved before the end of the job, will be deleted from the Directory when the job ends; such files are known as *temporary files*. Any file which outlives the job during which it was created is called a *permanent file*.

2.3 *Scope of the language*

The scope of the language is restricted in such a way as to achieve:

(*a*) closeness of the programming instructions to the basic operations executed by any operating system

(*b*) generality of instructions leading to a small number of powerful, simple and orthogonal programming concepts

(*c*) independence of particular operating systems or computing system architectures

(*d*) well structured programs. The semantics of a JOBOL program correspond to a simple composition of the semantics of its units (Hoare, 1972)

(*e*) provision for the relegation of system dependent parts of a JOBOL program to the string constants of the program

(*f*) provision of defaults for as many parameters as possible; the defaults are installation dependent. Transportability of JOBOL programs is achieved by considering whether the changes in defaults when going from one installation to another change the virtual environment for which the program has been written

(*g*) relative ease of implementation regardless of operating system, by exclusion of facilities which cannot be precompiled.

3. The programming language structures of JOBOL

A JOBOL program is a sequence of statements delimited by a start and a stop, the statements being separated by separator symbols. Each JOBOL statement is one of the following basic structures:

Procedure call
Assignment statement
Loop
Branch (Conditional statement)
Monitor.

3.1 *Procedure calls*

A procedure call invokes one of the standard procedures.

1. *RUN*

From a simplified viewpoint, this procedure loads an executable object program, including any linking involved, and executes it. This is what it looks like from the users' point of

view irrespective of whether it is, in fact, doing this, or is invoking a package of compiled or semicompiled modules, or is calling an operating system macro (e.g. an IBM catalogued procedure). The most frequent use of RUN will be to run either a compiler or a program compiled by such a compiler.

The principal operand of RUN is the name of the file containing the program or the package which is to be executed. Any executing program may input (or output) one or more linear sequences of data values. Each of these linear sequences of data values is called a datastream. Other operands of RUN set up a correspondence between each datastream used within the program and the JOBOL file to be used for input or output on that datastream. Alternatively, RUN may be followed by an installation-dependent parameter string.

2. *SAVE*
This extends the life of a file beyond the duration of the JOBOL program within which the SAVE is invoked. Such a file is of course a permanent file.

3. *GET*
This procedure obtains access to a permanent file.

4. *MACRO*
The MACRO call denotes a file whose contents is a sequence of JOBOL statements; the semantics of a MACRO call are that the call is replaced by the contents of the file.

5. *RENAME*
This replaces an existing file name by a different one. It is necessary because many systems, packages or programs may output to a file with a fixed name and this can cause difficulties if the package is to be used more than once in a single JOBOL program.

6. *ERASE*
This procedure destroys all directory entries relating to the file to be erased, so that the JOBOL file management system no longer has any record of it. Hence the file ceases to exist.

7. *REWIND*
This procedure resets the current position marker to the beginning of the file to be rewound.

1. *Alias Statement*
A file name in JOBOL programs is a reference to a file contents and to a file descriptor. The same file may be referenced at two distinct levels:

(*a*) within a JOBOL program (by a file name)

(*b*) within the level of nomenclature recognised by the operating system (by a system dependent string).

In order to specify, within a JOBOL program, that different denotations at different levels indicate the same file, an *alias statement* is made.

2. *File descriptor assignment*
A file descriptor is a data structure, associated with a file, containing a number of fields each of which is referred to by a TRIBNAME (file attribute name), and each file descriptor field may be assigned a TRIBVAL (attribute value). A Descriptor Assignment is the assignment of a set of TRIBVALS to the descriptor of the file denoted in the assignment statement. The values assigned may be any of the following:

(*a*) a simple copy of an existing descriptor

(*b*) a set of values, explicitly listed

(*c*) a copy of an existing descriptor modified by a list of values which are different.

3. *Contents assignment*
This reproduces the contents of an existing file as the contents of a new file; alternatively it may assign to the new file the concatenated contents of two or more files.

3.3 *Loops*
The loop in JOBOL provides a means of specifying repetition of certain operations a predetermined number of times, possibly using different files as operands in the course of the repetitions. The files to be used are specified in one or more loop control lists. Each control list uses a dummy variable (file identifier) as control variable, and has a list of actual file identifiers to be substituted, in turn, at each repetition. Therefore each control list of the loop contains the same number of actual file identifiers, and this is the number of times the loop will be performed.

3.4 *Branches*
The test on which branching occurs is the success or failure to match the contents of the specified file with the string supplied in a JOBOL conditional statement. This mechanism is very general and provides a means of intercommunication between different jobstep programs via the JOBOL program. It does not require any *particular* facilities or properties of the operating system to enable it to be implemented, and this is why the mechanism has been chosen. Furthermore, it has the virtue that no new types of variable or expression need to be introduced into the language in order to express the condition.

3.5 *Monitor*
This is the name given to a sequence of JOBOL statements which are to be executed if and only if the operating system would (in the absence of a monitor) take over control and abort the JOBOL program. Precisely what conditions would lead to an abort of this kind is installation dependent. The monitor facility makes it possible for the user to arrange, for example, to list files containing diagnostic information which he requires only in the event of an abort taking place.

4. JOBOL meets its objectives

The question now arises—does the design of JOBOL meet the objectives outlined? In particular, can the language be compiled without modification of the operating system and if so do its contructs map into existing job control languages?

The JOBOL procedure calls SAVE, GET, MACRO, ERASE, REWIND all correspond in an obvious way to operations existing in any system which has a file store. Of course the JOBOL implementation itself needs to do its own *book-keeping* in relation to SAVE, GET, ERASE. The RENAME procedure corresponds entirely to such book-keeping operations.

The implementation of RUN is, of course, loader dependent. Any executable file must be loaded by its intended loader. Such information will be stored in the File Descriptor for the file, and used by the JOBOL run time system routines.

JOBOL file descriptor assignments and alias assignment statements are again entirely machine independent since they amount to book-keeping. JOBOL Assignment Statements amount to copying operations from one file to another and depending on the attributes of the files involved, these may then be from disc to disc, from disc to magnetic tape or the reverse, from magnetic tape to magnetic tape, from input device to internal file, from internal file to output device, etc. Every operating system has facilities for carrying out these operations, whether they be system utilities, macros, or simply independent programs which have been added to the system. Appropriate routines are called by the JOBOL run time system.

The loop construct in JOBOL presents no special mapping difficulties since its specially restricted form ensures that every loop maps into a predetermined finite number of sequential statements.

4.1 *The JOBOL run time system*

A description of the implementation strategy for the JOBOL compiler is not a part of this paper, but some aspects of it must be mentioned in order to indicate how JOBOL language structures such as the BRANCH are mapped into existing job control languages.

The final output of the compiler consists of some form of machine dependent job control statements, some of which invoke the execution of members of the library of program modules constituting the JOBOL run time system.

The implementation of the BRANCH on a machine whose operating system does not specifically provide such a facility can be effected by provision of appropriate programs in the JOBOL run time system. In a run time system for a JOBOL compiler running under OS/360 the program MATCH is such a program. It compares the contents of a JOBOL file (called file 1 in the example which follows) with the string constant provided in the JOBOL conditional; it then sets the OS/360 *condition code* to a suitable non-zero value, *n*, unless the match succeeds.

```
//BRANCH    EXEC      PGM = MATCH
//DD1  DD   DSN = file1
//DD2  DD   *
string constant
/*
```

Subsequent jobsteps to be executed are given a condition parameter as follows:

(*a*) if MATCH fails

```
//EXEC PGM=nextjobstep,
COND=(n, EQ, BRANCH 1)
```

(*b*) if MATCH succeeds

```
//EXEC PGM=nextjobstep,
COND=(O, EQ, BRANCH 1).
```

In the run time system for a JOBOL compiler running under GEORGE 3, it suffices to have a program (called TEST) which reads the contents of the file specified in the JOBOL condition and uses the DISPLAY facility of GEORGE 3 to send it to the system monitoring file, e.g.

```
LOAD TEST
ASSIGN *ED,file1
ENTER
IF DISPLAY string constant  ,GO TO 2
commands to be executed if string constant fails
to match contents of file1
                .
                .
                .
GO TO   3
2 commands to be executed if match succeeds
                .
                .
                .
3 first command following end of branch
```

A JOBOL compiler running under the CDC operating systems SCOPE 3.4, SCOPE 2.0, or NOS/BE has a run time program, MATCH, which compares the contents of two files and generates a MODE error if no match is found. The following CDC control cards may be used to implement BRANCH:

```
MATCH (file1, file2)
control cards to be executed if match succeeds
SYS (NIL)
control cards to be executed if match fails
SYS (ALL)
control cards following end of branch
```

Effect of MODE error: skip to next SYS card.

Effect of SYS (NIL): if no error then skip to next SYS card.

Effect of SYS (ALL): if no error, ignore and continue; if met while skipping, restart processing. (ULCC, 1974)

The MONITOR construct is, as previously stated, operating system dependent. It can be implemented only if the operating system does provide facilities in certain circumstances for return of control from the system to a program. If it does, then these facilities may be utilised to implement MONITOR. If it does not, then this is regarded as a limitation of the system and not of JOBOL (just as lack of an online card punch would make it impossible to implement a JOBOL program requiring punched card output).

5. Syntax and usage of JOBOL

5.1 *The meta-language*

The syntax of JOBOL is concisely defined in an extended BNF notation. The extensions used are as follows:

1. Non-terminal names are printed in lower case.
2. Square brackets denote a single, optional occurrence of their contents.
3. Curly brackets denote repetition, zero or more times, of their contents.
4. Terminal symbols do not appear in the syntax. Instead, the *names* of terminal symbols appear. They are denoted by unbroken sequences of letters in upper case (of length one or more). Multiple space is equivalent to single space, which serves to terminate such a sequence, as it does in a natural language.

   The representations of all terminal symbols in the grammar are provided in a separate table. See Appendix.
5. Individual statements (productions) of the grammar are separated by semicolons.
6. The BNF symbol '::=' is abbreviated to '='.

5.2 *Formal syntax of Jobol*

See Appendix.

5.3 *Transportability*

In describing the effects of transporting a JOBOL job from one installation to another it is instructive to draw an analogy with the effect of transporting say an ALGOL (or FORTRAN) program from one compiler to another. The meaning of such an ALGOL program will be changed by two distinct factors:

(*a*) Hardware configuration

(*b*) Compiler defaults

An example of (*a*) is wordlength, and hence the precision of representation of floating point numbers. An example of (*b*) is the convention used by the compiler for rounding (in the course of evaluation of arithmetic expressions), e.g. truncation, or rounding-up, or rounding-even. The result of these differences is that a numerical calculation may well produce as many different answers as there are compilers on which it may be run. If the problem is ill-conditioned, the answers might vary by an order of magnitude. Of course, so that the program be run at all, there may be certain minor (or major) alterations necessary so as to conform with (for example) compiler dependent standard functions which differ from one compiler to another even in name.

When a JOBOL job is moved from one installation to another, it will be necessary to change certain (installation or compiler

dependent) parameter strings; in addition, the same JOBOL program may be expected to produce different results because different default options are standard at different installations. There are two kinds of JOBOL standards—universal standard names, and installation dependent standard defaults.

1. *Universal standard names*

(*a*) Names for Standard Device types, e.g.

LP	—line printer
TTY	—teletype
CR	—card reader
CPUNCH	—cardpunch
PTR	—papertape reader
PTPUNCH	—papertape punch
MT7	—magnetic tape, seven-track
MT9	—magnetic tape, nine-track

(*b*) Names for high level language compilers, e.g.

ALGOL60	BASIC	PASCAL
ALGOL68	COBOL	SIMULA
APL	FORTRAN	SNOBOL

A particular installation may have several compilers for a language, but one of them will be the compiler provided as the standard default (for the standard name of the language). Any others will have different, local names.

(*c*) Standard external datastream names

Within any program or package which requires one or more input datastreams and/or one or more output datastreams, a different denotation may be used to indicate each datastream. The precise form and particular representation for such a denotation varies from compiler to compiler, and may or may not be known to the user. The executing program may be a user's own (FORTRAN) program, or it may be a compiler. In either case there is some *external* datastream name which sometimes differs from, and sometimes is the same as, the datastream denotation within the program. For example, some FORTRAN compilers use the external datastream name TAPE6 to refer to the datastream denoted by '6' within a FORTRAN program.

When the program being used requires more than one datastream for input (or for output), the user may need to know the external datastream names which the program expects him to use. However, it often happens that only one is required (e.g. source input to a compiler). JOBOL provides standard universal datastream names for use in such circumstances by all programs at all installations. These include:

INSTREAM, OUTSTREAM, CODESTREAM, LIBSTREAM, LISTSTREAM, INFOSTREAM.

The first two of these are the JOBOL names for the datastreams normally input to and output by any user program. The third is the JOBOL name of the datastream used by a compiler to output its object or target code. LIBSTREAM is the JOBOL name for the program library required by a program when the RUN command is being obeyed. LISTSTREAM is the JOBOL name for the datastream used by a compiler to list the source program which is input to it. INFOSTREAM is the JOBOL name for the datastream used by a compiler for the output of error diagnostics and other information.

(*d*) Names for other standard files, e.g.

INPUT	—a file associated with the standard input device
OUPUT	—a file associated with the standard output device
RUNCODE	—a file associated by default with the CODESTREAM of a standard compiler or package
UNITREC	—e.g. card images

2. *Installation dependent standard defaults*

(*a*) Designation of a standard device type to be associated with certain standard files, e.g. Output, and Runcode.

(*b*) Designation of a particular compiler as the standard compiler for each programming language.

(*c*) Assignment of a certain set of file attribute values (tribvals) to the file descriptor of each standard file—see 5.4.

(*d*) Designation of certain strings or formats for strings which are used as parameter strings by compilers or packages, or used to provide scheduling, identification, resource allocation and accounting information following 'JOB.

5.4 *JOBOL File Descriptors*

The information required by a file management system is detailed, extensive and installation/environment dependent. Not all of it will be required by simple systems. The data structure we use to specify it will almost inevitably require modification to suit some sophistication which has been overlooked, or which has not yet been invented. This structure we call a FILE DESCRIPTOR. Its main fields are the FORMAT field and CONTROL field. These may be subdivided as follows.

Structure of a FILE DESCRIPTOR

FORMAT FIELDS

Possible field name	*Comment*
BLOCK	Block (or buffer) size in bytes.
BOBBIN	Physical identification of a removable unit of data storage (e.g. disc pack, tape reel, cassette, floppy disc).
BPI	Recording density in bits per inch.
BYTE	Number of bits per byte.
CODE	Code, e.g. BCD, ISO, BBCDIC, BIN, etc.
DEVICE	Actual hardware device type.
EOL	Code value of End of Line Marker used.
FILEID	Sequential identifying number (e.g. for one of many files written upon one magnetic tape).
LFV	Line (Record) length fixed or variable.
LINE	Maximum number of bytes per line.
ORG	File organisation—random access, sequential access, indexed sequential, partitioned, etc.
PARITY	Parity—odd, even or none.
STORE	Main store required (in addressable units, e.g. words or bytes).
TABS	Tabulation settings, for use when file is printed.

CONTROL FIELDS

Control over one or more of the various types of access to a file may often be required. Authorisation is usually recognised upon production of some kind of password, jobnumber, account code, etc. Any user who can name the appropriate password automatically obtains the corresponding access.

Possible field name	*Comment*
ADD	Permission to extend by writing on the end.
ALL	Permission for all types of access.
ALTER	Alteration of any field of the File Descriptor including control authorisation (passwords), name of file, etc.
DATE	Date (until which file is required to be

	saved).
LABEL	Standard label (if any).
ERASE	Permission to erase the File Descriptor, and hence lose the file.
WRITE	Permission to overwrite.
MULTI	Multiaccess permitted (concurrently) of each of the following types of access.
READ	Permission to read (i.e. copy).
RUN	Permission to execute the file contents.

We now give an example of a possible assignment (by an installation) of file attributes of the standard file UNITREC, expressed in the form of a Descriptor assignment:

```
'DESCR
 UNITREC = 'TRIBS  BLOCKSIZE    = (960)
                  ,LFV          = (F)
                  ,BYTEWIDTH    = (8)
                  ,PARITY       = (NONE)
                  ,PAGELENGTH   = (0)
                  ,LINEWIDTH    = (80)
                  ,ALTER        = (XYZBOSS)
                  ,DATE         = (1 JAN 80)
                  ,LABEL        = (NONE)
                  ,DEVICE       = (DISC);
```

5.5 *User files input as part of a JOB*

A job may contain not only a program but also (say) two files of data, each of the files being used by the program as a distinct input datastream. Such a file, preceded by a header, of form

```
'FILE identifier newline
```

and terminated by an end of file mark, of form

```
'EOF
```

may be inserted within a JOBOL program after all JOBOL commands (*cf.* CDC job control).

When only one file is inserted in a job, the identifier may be omitted from the header, in which case the standard file name INPUT is assumed by default.

5.6 *Examples of JOBOL programs*

1. To copy from standard input device to standard output device.

```
'JOB (<installation dependent parameters>)
'CONTS OUTPUT = INPUT
'FILE
<source to be copied>
'EOF
'EOJ
```

2. To copy from standard input device to magnetic tape.

```
'JOB (<installation dependent parameters>)
'DESCR MYTAPE3 = 'LIKE MT9C 'BUT BOBBIN =
                        (. . . . . ), 'FILEID = (3);
'CONTS MYTAPE3 = INPUT
'FILE
<source to be copied>
'EOF
'EOJ
```

3. To compile an ALGOL program from cards and run it with card data (at an installation using card reader as a standard input device).

```
'JOB (<installation dependent parameters>)
'RUN ALGOL60 'WITH INSTREAM = PROG;
'RUN RUNCODE 'WITH INSTREAM = DATA
'FILE PROG
<cards containing ALGOL 60 program>
'EOF
'FILE DATA
<cards containing data for ALGOL 60 program>
'EOF
'EOJ
```

4. To copy cards to output, where card code is BCD and local standard code is not BCD.

```
'JOB (<installation dependent parameters>)
'DESCR CARDS = 'LIKE INPUT 'BUT CODE = (BCD);
'CONTS OUTPUT = CARDS
'FILE CARDS
<cards to be copied>
'EOF
'EOJ
```

When JOB (1) is moved from installation, say, A, where the standard output device is a line printer, to installation B, where the standard output device is a paper tape punch, it will run successfully. However, at A the output will be printed on lineprinter paper, while at B, it will be punched on papertape.

When JOB (2) is moved, the physical identification (of the magnetic tape spool) which follows 'BOBBIN =' might have to be changed, or might remain the same, depending upon local installation requirements.

In the case of JOB (3), the ALGOL 60 compiler default options might be different at installations A and B, and the result of moving the job as it stands would be to receive such different options. If this is not what the user desires, he would need to insert an installation and compiler dependent string immediately following 'RUN ALGOL60, e.g.

```
'RUN ALGOL60 (M)
```

where, for a particular compiler, M asks for a symbolic program map.

5. To compile an ALGOL 60 program from cards and make the compiled version a permanent file called MYPROG.

```
'JOB (<installation dependent parameters>)
'RUN ALGOL60 'WITH CODESTREAM = MYPROG;
'SAVE MYPROG
'FILE MYPROG
<cards containing Algol source program>
'EOF
'EOJ
```

6. To run some compiled program stored in a permanent file called MYPROG, using two independent sets of data, one on cards and one on a permanent file called MYDATA, storing the output in two permanent files, RESULTS1 and RESULTS2.

```
'JOB (<installation dependent parameters>)
'GET MYDATA;
'GET MYPROG;
'FOR      IN  = INPUT      , MY DATA
'AND      OUT = RESULTS1   , RESULTS2
'DO 'RUN MYPROG
     'WITH INSTREAM  = IN
          ,OUTSTREAM = OUT
'REPEAT;
'SAVE RESULTS1, RESULTS2
'FILE
<cards containing data>
'EOF
'EOJ
```

7. To run a program called WAGEROLL, conditional upon the data having been updated, the updating being signified by the date having been written, in a predetermined format, to a file kept for this very purpose, called WAGEDATE.

```
'JCB (<installation dependent string>)
'GET WAGEDATE;
'IF WAGEDATE 'SAYS (26/10/84)
'THEN 'GET WAGEROLL;
      'RUN WAGEROLL
'FI
'EOJ
```

8. To compile an ALGOL 60 program on cards, producing a listing only if the compilation is unsuccessful.

```
'JOB (<installation dependent string>)
  'MONITOR
  'CONTS OUTPUT = LISTSTREAM, INFOSTREAM
  'STOP
'RUN ALGOL60
'FILE
<cards containing ALGOL 60 program>
'EOF
'EOJ
```

Acknowledgements

The author would like to thank various colleagues for many critical and fruitful discussions, but in particular Mr E. Nixon of the Department of Statistics and Computer Science, University College London, and Dr G. Loizou of the Department of Computer Science, Birbeck College, University of London.

Appendix

1. *The Syntax of JOBOL*

```
program   = JOB PSTRING [monitor] sequence {file }
            JOBEND;
sequence  = [statement {SEMICOLON statement}];
monitor   = MONITOR sequence STOP;
file      = FILE [FILEID] NEWLINE
            {file + FILECONTENTS} ENDOFFILE;
statement = DESCR FILEID {COMMA FILEID} ASSIGNOP
            descrexpression
          | CONTS FILEID ASSIGNOP FILEID
            {COMMA FILEID}
          | ALIAS FILEID ASSIGNOP PSTRING likepart
          | RUN FILEID [rundetails]
          | REWIND FILEID {COMMA FILEID}
          | GET FILEID [WITH passkeylist]
          | SAVE FILEID {COMMA FILEID}
          | MACRO FILEID
          | RENAME FILEID AS FILEID
          | ERASE FILEID {COMMA FILEID}
          | IF conditional {ELSEIF conditional}
            [ELSETHEN sequence] FI
          | FOR controlsequence {AND controlsequence}
            DO sequence REPEAT;
descrexpression
          = ON BOBBINTYPE BOBBINNAME [likepart]
            [BUT triblist]
          | likepart [BUT triblist]
          | TRIBS triblist;
likepart  = [LIKE FILEID];
triblist  = TRIBNAME ASSIGNTOP TRIBVAL
            {COMMA TRIBNAME ASSIGNOP TRIBVAL};
rundetails
          = PSTRING
          | WITH runpardef {COMMA runpardef};
runpardef = STANDARDSTREAMNAME ASSIGNOP FILEID
          | NONSTANDARDSTREAMNAME ASSIGNOP
            FILEID
          | STANDARDOPTION ASSIGNOP PSTRING;
passkeylist
          = passkey {COMMA passkey};
passkey   = KEYWORD ASSIGNOP PASSWORD;
conditional
          = FILEID SAYS PSTRING THEN sequence;
controlsequence
          = CONTROLFILEID ASSIGNOP FILEID
            {COMMA FILEID};
```

2. *Representation of terminal symbols*

PSTRING	(any string of characters with balanced parentheses)		
FILEID	LETTER {LETTER OR DIGIT}		
ALIAS	'ALIAS	IF	'IF
AND	'AND	JOB	'JOB
AS	'AS	JOBEND	'EOJ
ASSIGNOP	=	LIKE	'LIKE
BUT	'BUT	MACRO	'MACRO
COMMA	,	MONITOR	'MONITOR
CONTS	'CONTS	ON	'ON
DESCR	'DESCR	RENAME	'RENAME
DO	'DO	REPEAT	'REPEAT
ELSETHEN	'ELSETHEN	REWIND	'REWIND
ELSEIF	'ELSEIF	RUN	'RUN
ENDOFFILE	newline 'EOF	SAVE	'SAVE
ERASE	'ERASE	SAYS	'SAYS
FI	'FI	SEMICOLON	;
FOR	'FOR	STOP	'STOP
GET	'GET	THEN	'THEN
FILE	'FILE	TRIBS	'TRIBS
		WITH	'WITH

Each of the following may be represented by one of its own set of known identifiers (with the same representation rule as FILEID):

BOBBINTYPE KEYWORD STANDARDOPTION STANDARDSTREAMNAME TRIBNAME.

Each of the following may be represented by any identifier (with the same representation rule as FILEID):

PASSWORD CONTROLFILEID.

Each of the following may be represented by a PSTRING:

BOBBINNAME NONSTANDARDSTREAMNAME TRIBVAL.

FILECONTENTS is represented by any string not including ENDOFFILE.

References

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.

BARRON, D. W. (1974). Job Control Languages and Job Control Programs, *The Computer Journal*, Vol. 17 No. 3, pp. 282-286.

COLIN, A. J. T. (1971). *Introduction to Operating Systems*, Macdonald, London.

CONTROL DATA CORPORATION (1969). *Scope 3 Reference Manual*, (CDC Literature Catalogue number 60189400), Palo Alto.

FRANK, G. R. (1976). Job Control in the MU5 Operating system, *The Computer Journal*, Vol. 19 No. 2, pp. 139-144.

GOULD, I. H. (1971). *IFIP Guide to Concepts and Terms in Data Processing*, North-Holland.

HOARE, C. A. R. (1972). *Structured Programming*, pp. 83-174, 'Notes on data structuring.' Academic Press, New York.

HOWARTH, D. J., PAYNE, R. B. and SUMMER, F. H. (1962). The Manchester University Atlas Operating System, Part II: Users Description, *The Computer Journal*, Vol. 4, pp. 226-229.

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

INTERNATIONAL BUSINESS MACHINES (1967). *IBM System/360 Operating System Job Control Language*, (IBM systems reference library S360-48, C28-6539-4), White Plains.

MOORE, L. (1975). *The Feasibility of a Transportable Job Organisation Language*, M.Phil Thesis, University of London.

MORRIS, D. (1974). Job Control Languages, Past, Present and Future, *Job Control Languages* (Editor D. Simpson), NCC publications.

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

NEWMAN, I. A. (1975). Machine independent command language, *Computer Bulletin*, Series 2 No. 4, pp. 14-15.

NEWMAN, I. A. (1976). Machine independent command language, *Computer Bulletin*, Series 2 No. 9, pp. 18-19.

PALME, J. (1969). *What is a Good Programming Language?* FOA P Rappert C 8231-64 (11). Research Institute of Nat. Defense, Operations Research Center, Stockholm.

UNIVERSITY OF LONDON COMPUTER CENTRE. (1974). *Taking Alternate paths through Control Cards: SYS*. Bulletin B2.3/1.