A general control language : language structure and translation

R. J. Dakin

Euratom-UKAEA Association for Fusion Research, Culham Laboratory, Abingdon, Oxon, OX14 3DB

This paper develops the structure of a common interface language for accessing the facilities of a variety of operating systems. Major design criteria are the linguistic implications of job control specification and ease of translation to existing job control languages with a view to implementation on a satellite system connected to main frames via RJE links. The main features which emerge are a generalisation of function calls in which the parameter list constitutes a separate entity, the use of assignments of limited scope to provide a flexible means for handling options, a method for controlling the sequence of generated JCL based on chained strings and a variety of facilities which allow a user to modify the user image to suit his own requirements. The translator has been implemented on two computers and used to generate JCL for three target systems; the implementations appear to be sufficiently economical in resource requirements for satellite use.
(Received October 1973)

This paper introduces a General Control Language (GCL) which has been designed to provide a common means for expressing job control information when submitting jobs to a satellite computer for running on a main frame machine to which it is linked.

The term 'satellite' is taken to mean that it interacts with main frame systems via standard RJE interfaces so that, by and large, no special software for the main frame system is required. This implies that job control information (referred to later as JCL) must be generated from GCL by a Job Language Translator (JOLT) running on the satellite. Design implications of this mode of operation are that GCL cannot assume any close interaction with the target main frame system and that the language should be such that the storage and CPU requirements for JOLT are modest.

Such considerations have led to a strong emphasis on the development of a linguistic framework capable of dealing with the variety and changeability of job control information while retaining the structural simplicity necessary to allow economical translation—an emphasis that seems to be particularly necessary at present, since most available control languages appear to suffer from the lack of a truly appropriate structure.

This paper develops the structure of GCL and the main features of JOLT implementations rather than the actual user image which will form the subject of a later paper. Examples have been chosen with a job control flavour for illustrative purposes and do not necessarily reflect the currently implemented user image.

What emerges is a general translational system whose facilities resemble those of a sophisticated macroprocessor with a restricted input syntax but substantially greater run time efficiency and flexibility in the sequencing of generated text than a conventional macroprocessor. The scope of this system is potentially wider than satellite job control.

1. Main structural features

1.1. *Functional notation*

The notation of mathematical functions forms a convenient starting point for the development of GCL. It is very general and syntactically simple. Moreover, one tends naturally to think within a functional framework in the area of job control; thus running a program, compiling a source module or printing a body of text are all actions performed on objects—or functions performed on parameters.

While the adoption of functional notation, brackets and all, may not endear itself to all users, brackets have their advantages; for example nested statements such as

RUN(LINK(COMPILE(SOURCE)), (INPUT, F),
(OUTPUT, PRINTER))

cannot be constructed if we drop the brackets. Some users might consider this would be no great loss (indeed, GCL provides the means for avoiding nested statements) but, as we shall see, the use of brackets allows some powerful generalisation of concepts.

Nested statements become a good deal more readable if more than one type of bracket is allowed. In GCL the inequality signs ⟨⟩ can be used interchangeably with the normal bracket pair (); thus the above expression can be written

RUN(LINK⟨COMPILE(SOURCE)⟩, ⟨INPUT, F⟩,
⟨OUTPUT, PRINTER⟩) .

Note that in this example the correct sequence of operations:

compile
link edit and load
run program

is automatically followed if each parameter is evaluated before entering the function to which it applies (as in ALGOL call by value). This method of parameter passing is adopted for GCL.

1.2. *Generality and hierarchy*

Different operating systems frequently use different means to achieve similar ends. The translation from one target job control language into another is therefore likely to involve trying to infer from one JCL what the user intends, before expressing it in the other—a task which may be difficult or impossible.

To avoid difficulties of this nature, GCL seeks to be 'high level' in the sense that it allows the user to express what he intends to accomplish while remaining uncommitted as to ways and means. To the extent that this is achievable then one is left with the comparatively simpler task of mapping downwards on to ways and means without the need for any prior reverse mapping of the type implied by inferring the user's intentions.

To be realistic it seems over-ambitious to base our approach on the assumption that such an aim is achievable except for fairly simple tasks; if we are to stretch a target system's facilities to its limit some reflection of 'ways and means' in the GCL user image seems inevitable.

GCL provides the means to arrange the facilities of a given target system implementation into a hierarchy. High level facilities, which are substantially system independent, make use of lower level facilities which are more flexible and increasingly reflect the structure of the target system. Thus a user can employ simple system-independent statements where his requirements

allow it but can resort to less convenient, more system dependent, but syntactically similar statements where necessary. In terms of the function structure, facility hierarchy is reflected in function hierarchy in which a higher level function calls a sequence of lower level functions.

The mechanisms which allow the construction of a function comprising calls on other functions can, without additional implementation effort, be made available to users who can thus define their own functions to perform recurring tasks of similar form. The addition of input stream switching functions adds greatly to the value of such facilities; a session must start with a statement in which a user introduces himself to the system and the initialisation function can readily be made to invoke the stream switching function to cause the contents of a local file (whose name is determined from user identification) to be inserted into the input stream. Stream switching statements in this file can, in turn, cause further files to be inserted; this allows users to share function definitions, thus creating a GCL 'dialect' which, since it is invoked automatically by session initialisation, appears to be part of the system. This situation is saved from anarchy by the requirement that different dialects must map on to a common user image; it also has the unusual virtue of being oriented towards what the user wants rather than what a system designer thinks is good for him.

The use of hierarchically structured facilities seems to be particularly well suited to learning and teaching purposes. A user needs to know very little to get started—some simple structure and a few high level statements. As his requirements become more complex he can be progressively introduced to other high level facilities and lower level facilities. If the structure is well conceived it should be possible to avoid the substantial learning barriers which occur in many (all?) other job control languages—for example, when one first encounters a task which cannot be readily handled by available catalogued procedures in the IBM OS system.

1.3. *Options and defaults*

A comprehensive operating system generally requires a great deal of information from the user to allow it to allocate resources, make sensible scheduling decisions and provide those facilities that the user requires. It may also require a good deal of essentially redundant information to make its own task easier. To cut down on the amount of information a user must explicitly specify, the system can provide much of the information itself on the understanding that default settings are to be used for items which the user omits.

The notion of defaults is a useful one, but is not without its pitfalls when we come down to detail. First, there is the syntactic problem of how specific items and their settings are to be associated in a way that allows the user to omit items and, secondly, there is the problem of what the default settings should be. A set of defaults essentially defines a 'normal' mode of operation. If a particular user's requirements always differ from those defined by some 'standard' set of defaults then he must always override some defaults with identical settings whenever he prepares a job for presentation to the system. If default values are to be of maximum value then it should be possible to tailor them to a single user or group of users.

One solution to the syntactic problem is to extend normal functional notation to allow the omission of parameters which are defaulted. Thus

F(, A,·, B)

specifies that parameter 2 = A, parameter 4 = B and that parameters 1, 3, 5, 6, 7, . . . are to be defaulted. Its main drawback is that it is subject to clerical and memory errors when the number of parameters is large.

Another approach is to use key words rather than position to define the significance of parameters which can, then, appear in any order. For example

RUN(PROG=ANAL,INP=CARDS,OUTP=PRINTER)

By this means anything corresponding to an absent keyword can be defaulted. This seems to be a cleaner way to handle default specifications but is irksome for parameters which cannot be defaulted.

GCL seeks to get the best of both worlds by making a clear distinction between 'parameters', which are positionally defined and specify information which must be present, and 'options', which are keyword defined and can be defaulted. Thus in a function call, a parameter list may be followed by an option list, the two being separated by a colon within the parameter brackets. For example:

RUN(ANAL, ⟨INPUT, CARDS⟩, ⟨OUTPUT, PRINTER⟩ :STORE = 200);

An important question is whether the names of options which are used in this way should 'belong' to a particular function or whether they should be globally defined. The use of global option names may impose some load on the implementor's imagination and the user's memory in cases where similar but distinct concepts arise in different functions, since distinct names must be used; such difficulties have not been great in practice to date. Localised option names, on the other hand, seem to have overwhelming disadvantages in the context of function hierarchy.

Suppose, for example, that an option called STORE belongs to the function RUNB; a function RUN which calls RUNB will, in all probability, make the same option available to users: this is most straightforwardly accomplished by creating an option, also called STORE but belonging to RUN, which has the same significance as RUNB's STORE and is explicitly handed down by RUN when it calls RUNB. The awkwardness of this explicit handing on of options is, I trust, evident, especially if there are other options called, say, TLIM, PRIORITY, . . . in the same category as STORE; but the real crunch comes when we consider another function called CATALOGUE, say, which also has an option called STORE which is, however, semantically distinct from RUNB's STORE. If RUN now calls CATALOGUE as well as RUNB, it cannot make both RUNB's and CATALOGUE's option STORE available in the straightforward manner described above. Renaming one of them seems to be a recipe for confusion, but if we insist that a given option name be tied to a particular semantic significance we might just as well use global option names. No other solution will, in general, avoid this impasse if we are to allow users to define their own functions.

Thus, in GCL, option names are globally defined. This makes it possible to allow option settings to appear as separate statements which have the effect of resetting the default value. For example, the two statements

STORE = 200

followed by

RUN(ANAL, ⟨INPUT, CARDS⟩, ⟨OUTPUT, PRINTER⟩)

cause the same action as the single statement in the previous example but with the additional effect that the default value of the STORE option is still set to 200 when executing subsequent statements. Since the default setting statements associated with a particular user and/or group of users can be automatically invoked at session initialisation by the means described in section 1.2, the second of the requirements which we laid down for default specification has been met.

1.4. *Commenting and layout*

Control languages, no less than any other computing languages, should provide freedom of layout and the ability to add com-

ments to allow one to produce a readable document. To this end GCL adopts the following rules on input.

(*a*) A line may contain more than one statement or a single statement may span several lines.

(*b*) The slash character (/) acts as logical end of line—allowing comment to follow it. Note that comment may be inserted in the middle of a statement by this means.

(*c*) The space character is used, where necessary, to delimit elements such as identifiers and integers and is significant in character strings. Otherwise spaces are insignificant.

(*d*) A GCL statement is terminated by a semicolon.

Example

```
STORE = 200;/(RUN WITH 150 GAVE
                MEMORY VIOLATION)
RUN(PROGFILE⟨'REDUCE'⟩,/COMPILED 17/1/73
                ⟨INPUT, DATA⟩, / FROM CARDS —
                                COMPLETE UP TO 30/6/73
                ⟨OUTPUT, PRINTER⟩);
```

GCL also includes the ability to accept bodies of source and data text, in line with control statements, in which text is accepted in a transparent manner and hence the above rules cannot apply. A truly global rule is, however, applied to all input lines:

(*e*) Any line commencing with the two characters + + causes the remainder of the line (and following lines if the statement spans more than one line) to be interpreted as a GCL statement and executed immediately. After executing this statement the translator normally discards the rest of the line on which the statement terminates and reverts to whatever activity it was which caused the + + line to be read.

The purpose of this last facility is twofold—it allows one to invoke channel switching facilities while reading text, thus effectively replacing the + + statement by the contents of a file and, secondly, it provides a means of protecting one user from the errors of the preceding one in a system in which succeeding sessions follow each other in a single stream. If the session initialisation statement is in a line commencing with + + then it is certain to be executed, regardless of whether or not the previous session was correctly terminated, thus allowing appropriate action to be taken if it was not.

Should the + + marker prove to be inconvenient (because it occurs naturally in text) then some other string could be selected. Under circumstances where no character string can be regarded as sacrosanct then the immediate execution marker could be made a user option, but at the cost of losing some of the protection against previous user faults given by a fixed marker system. This does not matter if separate sessions are in separate streams of input, where premature termination of the input stream can be taken to indicate the absence of proper session termination.

1.5. *Primitive functions*

At the base of the function hierarchy are 'primitive functions' which do not comprise calls on other functions but form an integral part of the translation system. A feature of these functions is that they are quite independent of any target system; in fact they are indepencent of any job control concepts whatsoever, so that the basic translation system is not necessarily limited to job control applications.

There are currently 21 primitive functions which are summarised in Appendix 2. Two groups of primitives are of particular interest—text generating primitives and control primitives.

1.5.1. *Text generating primitives*

The generating of different target JCL from a common GCL involves sequencing problems which rule out a simple linear generation of output text. For example some systems (such as the IBM OS system) require JCL to be interleaved with source code and data, while in others (such as CDC SCOPE) essentially all JCL for a session precedes source code and data. There are many more detailed differences in structure which could be cited; it suffices to say that, in general, a single GCL statement may give rise to the insertion of JCL at more than one place.

The necessary flexibility has been achieved by generating JCL in the form of a forward chained list of character strings which we shall call the 'output chain'. Each entry in the list comprises a header, consisting of a string count and a pointer to the next entry in the list packed into one word, and up to eight pointers to strings. Any number of pointers into the output chain may be retained, allowing insertions to be made between any two previously generated entries. The primitive function which generates an entry is called GEN and allows one to eight strings to be inserted following a specified entry.

For the sake of efficiency the output chain is retained in primary storage, but the strings themselves could be kept on secondary storage without substantially degrading performance. The scheme is, however, quite economical in string storage since repetitions of a string in the output chain involve repeated pointers to the same string.

Source and data text are buffered in local (satellite) files; their positions in the output chain are indicated by special strings, as described in Section 2.3.

1.5.2. *The IF and LOOP primitives*

In most languages conditional and repetitive operations are expressed by special syntactic forms. In the interests of implementation economy, GCL sticks to functional notation, but at some cost in clarity. Since most users should only use IF and LOOP functions indirectly via higher level functions, this loss of clarity does not appreciably affect the user image.

For both functions the first parameter is an integer which controls the evaluation of subsequent parameters, which are GCL functions. The notation is simplified by allowing query (?) as a parameter separator which is equivalent to enclosing the following parameter in function brackets (see Section 2.6).

The IF statement interprets the first parameter as a truth value with the convention that *true* and *false* are represented by the integers 0 and 1 respectively, which cause the second or third parameter, respectively, to be evaluated and returned as the value of the IF statement. There can be only two parameters, in which case a first parameter value of 1 (false) causes a null value to be returned. In any case, parameters which are not selected by the first parameter value are not evaluated, so that any resulting side effects do not occur. The first parameter can be greater than 1 in which case it generalises to a species of case statement which allows the selective evaluation of one of any number of functions.

Examples

```
IF(EQ⟨A, 3⟩? FAIL⟨15⟩;RETURN⟨⟩);/FAIL 15 & EXIT IF A = 3
IF(N?'ZERO'?'ONE'?'TWO');/RETURN NAME OF N's VALUE
```

The LOOP statement interprets its first parameter as a repetition count; the second parameter is then evaluated this number of times. The asterisk symbol (*) appearing in the second statement is the current cycle count. In other contexts * has a different significance as we shall see.

Example

```
LOOP⟨3? F(*)⟩;/EXECUTE F(1), F(2) & F(3)
```

1.6. *Named constants*

Internal conventions such as the true/false representations in the previous section can be masked from the user by attaching names to constants. Named constants also help to safeguard system integrity.

For example, the integer constants 0 and 1 can be represented by the names YES and NO respectively, allowing option settings such as

```
REFERENCES = YES;
```

A function body might, then, access the REFERENCES option in a statement such as

```
IF⟨REFERENCES?GEN(ATCUR,',MAP')⟩;
```

1.7. *Identifiers, declarations and assignments*

Since function names, option names and constant names are all global in extent, they are accessed via a single dictionary in JOLT. Primitive functions and other predefined entities are associated with identifiers in a more or less fixed manner; further identifiers are declared and associated with higher level functions, options and constants in 'mapping statements' which are syntactically identical to user statements.

Thus users can introduce their own identifiers if required. Substantial developments along these lines could, however, be hindered by the fact that identifiers are globally defined. An individual user would find it difficult enough to avoid identifiers which occur in a growing list of reserved system names, while a co-operating group of users would face the additional problem of mutual clashes.

GCL goes some way towards meeting this problem by adopting naming conventions. System identifiers are completely alphabetic and at least two characters long. An individual user can avoid clashes with system names by using single character identifiers or multiple character identifiers containing numerics or the ampersand (&) symbol, which is treated as an alphabetic letter. Groups of co-operating users could adopt further conventions to avoid mutual name clashes. Given an adequate user image defined by system mapping statements the need for superimposing user facilities should be minimal, and the above conventions should provide an adequate framework. This position may need to be reviewed in the light of experience. The implementation of nested scopes would be quite costly in terms of satellite resources and may not solve all the problems, but one can envisage less drastic changes which would improve the situation.

GCL avoids the need for separate declaration statements, while retaining the protection against mis-spellings and similar errors afforded by declarations, by insisting that an identifier must be preceded by an exclamation mark (!) the first time it appears in the text. On being declared in this way it is initialised to a null value but may subsequently be associated with any other entity via assignment. References to undeclared identifiers or multiple declarations of the same identifier will generate failures. An identifier is defined at all times subsequent to its declaration. It can be declared in any GCL statement.

An assignment is a construct of the form

⟨identifier⟩ = ⟨expression⟩

(where the symbols ⟨⟩ are here used as meta brackets in the usual BNF manner—see Appendix 1). The effect of an assignment is limited in scope, to provide the required option default and override behaviour, by the following rule:

> An assignment is effective only while executing the function or statement in which it occurs; thereafter the identifier reverts to its previous setting.

Thus, an assignment in an option list is effective only while executing the function called. An assignment which occurs as a separate statement inside a function is only effective while executing the function. Outside of function bodies separate assignments have unlimited scope.

Apart from the required option default override behaviour, this arrangement goes a long way towards providing local variable facilities in the language. Thus one can use the identifier TEMP, say, to hold intermediate results in any number of functions without risk of mutual interference. It differs from a set of truly local variables in that it is declared only once and has a well defined initial value (see Section 2.1). A number of identifiers can be reserved for local use by all functions.

Limited assignment scope makes it impossible to write a function which uses assignments to initialise identifier values, since any assignments inside the function are nullified on exit. To overcome this limitation an alternative form of assignment with unlimited scope is provided by the SET primitive function. Note however that the effect of a SET will be nullified once control goes outside the scope of a previous ordinary assignment to the same identifier.

Examples

```
!LANGUAGE = FORTRAN;/SYSTEM OPTION
/     'LANGUAGE' SET TO DEFAULT OF FORTRAN

!&MYNAME = 'DAKIN';/USER IDENTIFIER
/     &MYNAME INITIALISED TO 'DAKIN' .
```

2. Data types

We have so far encountered entities as disparate as integers, character strings and functions which can be associated with identifiers. Each of these (and a number of others) are regarded as GCL data types. GCL gains a great deal of flexibility, while retaining adequate means for internal checking, by associating data type with values explicitly and allowing the type associated with an identifier to vary dynamically at run time.

Internally a value is represented by two words which represent quantity and type. For integers, the quantity is the internal (binary) representation of the integer itself. For more complex values the quantity is a pointer to an appropriate data structure which represents the value. The type of a value can be accessed in GCL expressions using the TYPE primitive.

A constant may be of any type; the fact that it is a constant is indicated by a flag in the type word of the identifier value which causes any assignment to the identifier to generate a failure. The constant flag is not passed on when an identifier is accessed but must be generated as part of a constant setting by the primitive function CONST.

For example:

```
!YES = CONST(0);/YES IS THE CONSTANT 0
!A = YES;          /USER IDENTIFIER A INITIALISED
/                  TO 0 (NOT CONSTANT)
```

The complete set of GCL data types follows. Most have an explicit form, others are predefined and preassigned to particular identifiers. All are 'first class citizens' in the sense that they can be assigned to identifiers, returned as the result of function evaluations, passed on as call parameters and generally accessed in exactly the same way.

2.1. *Null*

This type provides a means of indicating that nothing has happened. Thus, an identifier is set to a null value on being declared, and can be returned as the result of function evaluation to indicate that no value is returned (there being no other formal distinction between functions and procedures). There is no explicit form other than a newly declared identifier.

2.2. *Integer*

GCL allows unsigned integers only. The explicit form is a string of digits. Several examples have appeared already.

2.3. *String*

Character strings are used to represent information destined for insertion in the output chain. The explicit form is a set of characters enclosed in primes (').

Inside an explicit string slash retains its significance of line terminator (allowing a string to span several lines) and semicolon retains its significance as statement terminator (allowing the detection of unterminated strings—an error which might otherwise cause a host of irrelevant failure indications). The asterisk has an overriding effect which causes the character which follows it to be accepted, willy nilly, as part of the string and interpreted in a particular way (a very useful device borrowed from CPL). Thus, inside a string,

*	is represented	by ** ,
'		by *' ,
/		by */ ,
;		by *; ,
new line		by *N and
buffered text marker		by *F.

The effect of *F on output is to cause the remainder of the string in which it occurs to be ignored and the following string to be expanded into the name of a local file containing buffered text (as set up by the OPENOUT and COPYTO primitives).

2.4. *List*

At a small additional cost in terms of implementation the parameter passing mechanism of JOLT has been generalised to allow what has been so far called a parameter list to constitute a separate data type. The explicit form of a list is that of a parameter list. Thus, a list can be assigned to an identifier and can itself be a list element. Lists provide a powerful structure building mechanism and constitute the principle means for creating data structures to represent such job control concepts as devices, libraries and the like.

Example

```
RUN(ANAL, CARDS, PRINTER);
```

is equivalent to

```
!&MYJOB = (ANAL, CARDS, PRINTER);
RUN &MYJOB;
```

Elements of a list can be indexed via constructs of the form

⟨list⟩ (⟨index⟩)

where ⟨index⟩ may be an expression returning a value of type integer. The index parentheses can be dropped where no ambiguity results. A zero index is allowed and returns an integer value equal to the length of the list.

Example

```
!L = ('ONE', 'TWO', 'THREE');
L 2; L(2);/BOTH RETURN 'TWO'
!N = L(0);/SETS N to 3 .
```

2.5. *Primitive function*

A primitive has no explicit form as it constitutes an integral part of JOLT. It can, however, be handled in the same way as other values. A function itself is distinguished from an evaluation of the function by the presence or absence of call parameters; if no call parameters are required the null list () is used to cause evaluation. For implementation convenience the IF, LOOP and VAL primitives are treated internally as separate value types.

Example

```
!A = ADD; /ASSIGN ADD PRIMITIVE TO A
!S = A(B, C); /SET S TO B + C .
```

2.6. *GCL function*

The explicit form of a function is a sequence of statements enclosed between an initial @ and a terminating # character. The semicolon terminator on the last statement in a function is optional. No special exit statement is required at the end of the function, but conditional exit from the middle of a function can be effected by use of the RETURN primitive. The value returned by a function is the value generated by the last statement in the function or the RETURN call parameter. A function is only evaluated when it is applied to a parameter list or acted on by the IF, LOOP or VAL primitives.

The first, second . . . call parameters can be accessed as $1, $2, . . .; $0 represents the number of call parameters. The parameter list is automatically assigned to the identifier PARAM on entry, allowing the more general list accessing mechanism to be employed for parameters (useful for sequential access via the LOOP primitive) and permitting the entire parameter list to be handed on to a lower level function.

Example

```
!DOUBLERUN = /FUNCTION ASSIGNED TO
                DOUBLERUN
@ / $1          IS THE DATA,
/ $2            IS THE PROGRAM FOR THE
/               PREPROCESS RUN
/ $3            IS A PROGRAM WHICH USES
/               PREPROCESSED DATA
/               TO PRODUCE THE FINAL RESULTS
RUN ($2, $1, TEMP); /PREPROCESS
RUN ($3, TEMP, OUTPUT); /MAIN RUN
OUTPUT # / RETURN OUTPUT
;/  (TERMINATES ASSIGNMENT TO DOUBLERUN)
/ EXAMPLE OF FUNCTION CALL:
PRINT(DOUBLERUN⟨DATA,REDUCE,ANALYSE⟩);
```

When a function is an element of a list the notation can be simplified, as described in Section 1.5.2.

2.7. *Output chain pointer*

There is no explicit form, a single pointer being pre-assigned to the identifier CURPTR by the system. Thereafter any number of different pointer values can be retained by assignment at appropriate times. A call on the GEN primitive updates a specified identifier to point to the output chain entry which is generated.

3. Some generalisations and uses of GCL structure

3.1. *Extended form of GCL expressions*

The GCL expression

F(3)

could represent function evaluation if the value of F is of type function (or primitive function), but if F is a list it would represent the third element of F. If F is of any other type the expression is invalid. It follows that it is impossible in general to distinguish between function evaluation, list indexing and invalid expressions by syntactic means. In any case we say that F is 'applied' to (3), with the understanding that it might represent an illegal application which is detected at execution time. Thus, a GCL expression consists of either a single element (an element being an identifier, call parameter or the explicit form of one of the data types) or one element applied to another.

This has been generalised to allow an expression to consist of a sequence of any number of elements, evaluated by applying the first element to the second, the result of this application to the third, and so on.

Example

```
!FORTRAN = CONST (1);/LANGUAGE INDICES—
/                                    FORTRAN
!ALGOL = CONST (2);/                      —ALGOL
!LANGUAGE = FORTRAN;/DEFAULT LANGUAGE
!COMPILE = CONST@/FUNCTION TO GENERATE
/                                          JCL
/TO COMPILE A SOURCE MODULE (SUPPLIED AS $1).
/FCOMP AND ACOMP ARE PREVIOUSLY DEFINED
/                                    FUNCTIONS.
/WHICH GENERATE JCL FOR FORTRAN AND ALGOL
/COMPILATION RESPECTIVELY.
(FCOMP, ACOMP)LANGUAGE($1);
                              /SELECT APPROPRIATE
/  FUNCTION AND APPLY TO /$1
#; /END OF COMPILE
```

Application can be generalised to other data types, where this is meaningful and useful. One such generalisation has been implemented: the characters of a string can be indexed, thus allowing a string to be checked for syntax.

Example

```
IF(GT⟨STRING(0), 6⟩ ? FAIL⟨99⟩); /FAIL 99 IF STRING
/                                    LONGER THAN 6
```

Another generalisation of dubious value has been made—to allow the dropping of parameter brackets where this can be done without ambiguity. Thus, if F is a function, F 'AB' has the same effect as F('AB'). This can be dangerous at user level, since a user may not know that, say, files are represented internally by lists; if A is a list then F A is *not* equivalent to F(A). The facility has been used in constructing mapping statements but is not described in user documentation.

3.2. *Option hierarchy*

We have previously discussed the advantages of structuring GCL facilities in a hierarchy. In terms of language constructs these considerations apply no less to options than to functions.

A typical example of the need for option hierarchy is provided by 'streams' in the Multijob operating system. A stream is a fixed partition of main storage. Each stream may or may not be rolled in and out from secondary storage (mainly for interactive purposes) and is associated with specific I/O devices; such characteristics are fixed when a particular system is configured. Most remote users should not need to know about streams and configuration details except insofar as they affect available storage and other resources. Mapping functions should automatically select a stream appropriate to user requirements. Nevertheless, a user with unusual requirements not catered for by the mapping functions should be allowed to specify a particular stream directly.

GCL option hierarchy is implemented in the following way. Low level (system dependent) options, such as Multijob stream specification, are set to default values which are functions which would normally involve higher level options. When a low level option is accessed, the VAL primitive is applied to it to evaluate the function and thus generate the option value appropriate to the higher level option settings which are then current. The VAL primitive can, however, be applied to values of type other than GCL function, in which case it simply returns the value itself; thus a user can directly override a low level option with a specific value.

Example

```
!STREAM = @ IF ⟨INTERACTIVE? 'B'?'E'⟩#;/
/  DEFAULT-INTERACTIVE RUNS IN STREAM B,
/  OTHERWISE E
GEN(ATCUR,VAL⟨STREAM⟩);/EXAMPLE OF ACCESS
/  TO STREAM OPTION
RUN(PROGRAM,DATA,PRINTER:STREAM = 'A');/
/  EXAMPLE OF LOW LEVEL OPTION OVERRIDE
```

3.3. *Avoidance of nested statements*

The provision of user declared identifiers allows a user to select between nested function calls and a more step-wise approach, according to taste. Thus the actions specified by

```
RUN(LINK⟨COMPILE(SOURCE)⟩,⟨INPUT,CARDS⟩,
                              ⟨OUTPUT,PRINTER⟩);
```

could be expressed by the statements:

```
!R = COMPILE(SOURCE);
R = LINK(R);
RUN(R,⟨INPUT,CARDS⟩,⟨OUTPUT,PRINTER⟩);
```

A more convenient, though less general, means for achieving the same end is provided by the 'refer-back' symbol asterisk (*) which stands for the value returned by the previous statement. Thus, the above example could be expressed by the statements:

```
COMPILE(SOURCE);
LINK(*);
RUN(*,⟨INPUT,CARDS⟩,⟨OUTPUT,PRINTER⟩);
```

Note that asterisk is used somewhat differently in the second parameter of the LOOP primitive, as described in Section 1.5.2.

3.4. *User information facilities*

Layout and commenting facilities can be used in the mapping functions just as in user statements. The text of mapping functions can, then, be made to form a readable document which is made available to users in some form—for example by selective interactive interrogation of a file containing the text. Since this text contains the information that is actually used to set up the translation it is automatically up to date—a most unusual feature in user documentation.

The idea of rigidly linking user information to the system itself was suggested to me by Mr D. E. T. F. Ashby of Culham Laboratory. An example of the type of user documentation which can be produced in this way is given in Appendix 3, which is a section of the mapping functions for the ICL Multijob System.

4. Implementation

The JOLT job language translator for GCL has been implemented for two machines—the ICL 4-70 and the CTL Modular One. These comprise essentially a single implementation, as we shall see. Its main features will now be described.

4.1. *The identifier dictionary*

Identifiers are kept in a dictionary of three word entries, each comprising a pointer to the identifier string and its current (two word) value. The dictionary is accessed by hash coding the identifier and using Hopgood and Davenport's quadratic search method (Hopgood and Davenport, 1972) to resolve clashes; this technique requires a dictionary capacity which is a power of two. The need for precomputed hash tables is avoided by insisting that a set of mapping functions commence with a set of statements, each declaring a single identifier that is to be initialised to a primitive function or other preset value in a predetermined sequence. These are read and alloted dictionary slots via the normal compilation mechanism. In this way, primitive function and other preset identifiers are not integral to JOLT but can be altered by changing the mapping functions.

4.2. *Statement execution*

Each statement is first compiled into an internal form and then interpreted. The internal form—in which all data types have been already stored and identifiers have been replaced by

pointers into the dictionary—can be interpreted rapidly. Since function bodies and literal expressions are stored internally in this form, a highly nested function structure can be executed much faster than would be the case if statements were directly interpreted from their character representation as in most macroprocessors. JOLT could be regarded as a species of macroprocessor in which calls on lower level macros are invoked via direct internal linkages to the lower level macro bodies rather than via the generation and interpretation of macro calls in character form.

Since GCL structure can be nested, both the compiler and interpreter are used recursively.

4.3. *Store organisation*

The general approach to store organisation for JOLT is to avoid the need for complicated garbage collection mechanisms by using stacks (themselves a simple form of garbage collection) where possible and otherwise economising on off-stack storage.

The storage of lists and function bodies provides a particularly interesting case. While parameter lists and functions which comprise list elements are most economically stored in a stack, their elevation to first class citizenship creates problems since they can persist after the stack has been overwritten. The placing of all lists and function bodies in permanent storage would, in the absence of garbage collection, be very expensive and is in any case quite unnecessary where lists are used simply for parameter passing. The JOLT implementation adopts a composite solution.

Lists and function bodies are always stacked when they are initially formed. If a stacked list or function is assigned it is automatically copied to permanent storage. Since stacked list elements can themselves be stacked lists or functions, the copying process is recursive.

Nested assignment scopes are implemented by holding a reset stack of dictionary pointers and pre-assignment values. Wherever the interpreter finishes evaluating an expression the stack level is restored to its value before the evaluation commenced and any intervening resets are performed in reverse order (to cope with multiple assignments to the same identifier). An assignment always causes a new reset entry to be generated before the assignment is performed. The SET primitive does not generate a reset entry.

4.4. *Implementation language and portability*

JOLT has been implemented in the CLSD language (Calderbank and Calderbank, 1973), which provides convenient and efficient recursive and pointer facilities which make it very suitable for this purpose. The design of CLSD allows it to be macroprocessed into assembly code via the Stage 2 macroprocessor (Waite, 1970). CLSD can be implemented on a new machine in under three man months.

JOLT was first implemented on the ICL 4-70 running under Multijob and later moved to the Modular One, CLSD macros being already available for both machines. The time taken to move it to the Modular One, at the same time writing and testing a number of utility routines and text buffering facilities not previously included in the 4-70 version (about 120 additional CLSD statements), was under three man weeks.

The Modular One version requires 3,500 16-bit words of code and a variable amount of data storage, depending on mapping functions and user statements. Some 4,000 words of data are required to map a somewhat limited GCL user image on to JCL for the IBM OS system. This is an early version of JOLT which does not include input stream switching (which would not be practicable with the limited filing system used). The Multijob version includes all the facilities described.

4.5. *Mapping functions*

Three sets of mapping functions have been developed, producing JCL for the ICL Multijob operating system at Culham, the ICL George 3/4 system, and the IBM OS system, accessed via HASP, at the Harwell Atomic Energy Research Establishment. The user image itself will be described in another paper, but some general observations are appropriate here.

The Multijob mapping functions are by far the most comprehensive. They include facilities for compiling, link editing and running programs written in FORTRAN, ALGOL, COBOL, Assembly Code and CLSD as well as facilities for creating, updating and using private subroutine libraries, and a number of file manipulation facilities. The facilities for compiling CLSD are particularly interesting in that they are not integrated into the Multijob operating system as are the other compilers. This fact is not, however, apparent in the user image: the option setting LANGUAGE = CLSD causes the entire sequence of preprocess, macroprocess and assembly code compilation to be automatically invoked as required. Thus one can envisage the development of systems with a minimum of integrated 'system' facilities which nevertheless can present a sophisticated user image by using JOLT, or something like it, to invoke appropriate utility routines.

The OS mapping functions produce, for the main part, calls on catalogued procedures. Since catalogued procedures are limited to a single level and therefore rather specialised in function, this approach does not lend itself readily to a well-structured hierarchical GCL implementation but presents a useful subset of facilities and has the advantages of rapid implementation and minimal maintenance, since any changes in local practice tend to be incorporated in catalogued procedures and thus require no change to the mapping functions.

The George implementation caters for FORTRAN programs only, but is otherwise fairly comprehensive.

The Multijob version is available to users at Culham, but while it is simply an alternative to Multijob JCL, which most users have already mastered, it has been slow to catch on. We can expect GCL usage to accelerate once it becomes available as a truly general control language within a satellite system. User reactions have so far been generally encouraging.

5. Conclusions

The GCL structure which has been described seems to go a long way towards meeting the objectives of a general control language which presents an acceptable user image and can be implemented within the resources of a satellite computer of moderate size and power.

Acknowledgements

In the course of this work I have had many discussions which have given rise to valuable comment, criticism, suggestions and encouragement. I am particularly indebted to Dr. M. D. Poole (now at Oxford Regional Hospital Board), Mr. D. Ashby, Dr. M. Calderbank and Dr. T. Lang for their comments. Dr. Poole also started the Stable User Image project at Culham. An internal report by him, which surveys the job control area, provided a useful starting point for this work. Mr. Ashby also carried out preliminary investigations of user requirements at Culham. Mr. N. Risidore, a student from Brunel University, implemented a number of improvements and extensions to the original version of JOLT, and extended the Multijob mapping functions to cover ALGOL, COBOL, Assembler and CLSD compilation. Another sandwich course student, Mr. G. Benson from North Staffs. Polytechnic, constructed the George mappings; Mr. Benson is attached to ICL who supported him during his work at Culham.

The design of GCL has been influenced in a general way by the work of Jackson (1970) and Stoy and Strachey (1972).

Appendix 1 Formal syntax

The syntax shall be expressed in the following version of Backus-Naur notation.

Syntactic constructs are denoted by lower case words inside the meta brackets ⟨⟩. Repetition of a construct is indicated by a superscript asterisk (0 or more repetitions) or circled plus sign (1 or more repetitions) following the construct. A sequence of constructs to be considered as a single unit for repetition or when specifying alternatives is enclosed in the meta brackets {}. For clarity, we shall take as read the conventions that spaces can act as delimiters but are otherwise insignificant outside strings, that the slash character acts as logical end of line, that physical end of line terminates comment but is otherwise without significance in statements and that ++ at the start of a line indicates immediate execution regardless of context. Only statement syntax is specified, there being no larger syntactic unit.

⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|
U|V|W|X|Y|Z|&
⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9
⟨identifier⟩ ::= ⟨letter⟩⟨letter or digit⟩*
⟨declaration⟩ ::= !⟨identifier⟩
⟨integer⟩ ::= ⟨digit⟩ ⊕
⟨formal parameter⟩ ::= $⟨integer⟩
⟨special string character⟩ ::= *|'|;|/ . .
⟨string element⟩ ::= ⟨any character except a special string character⟩|*F|*N|**|*'|*;|*/
⟨string⟩ ::= '⟨string element⟩*'
⟨element⟩ ::= ⟨identifier⟩|⟨declaration⟩|⟨integer⟩|⟨formal parameter⟩|⟨string⟩|*|⟨list⟩|⟨function⟩
⟨expression⟩ ::= ⟨element⟩⊕
⟨assignment⟩ ::= {⟨identifier⟩|⟨declaration⟩} = ⟨expression⟩
⟨statement⟩ ::= ⟨expression⟩;|⟨assignment⟩;
⟨function⟩ ::= @⟨statement⟩⊕#
⟨list separator⟩ ::= ,|?
⟨list body⟩ ::= ⟨null⟩|⟨expression⟩{⟨list separator⟩⟨expression⟩}*
⟨option list⟩ ::= ⟨null⟩|, ⟨assignment⟩{⟨,assignment⟩}*
:⟨assignment⟩{,⟨assignment⟩}*
⟨list⟩ ::= (⟨list body⟩⟨option list⟩)|
⟨⟨list body⟩⟨option list⟩⟩

Implementation restrictions

The current implementation imposes the following restrictions on the above syntax:

1. An identifier may not be more than 12 characters long.
2. An integer i must lie in the range
$$0 \leq i \leq 32767 .$$
3. A list cannot contain more than 127 elements (excluding the option list).
4. An option list may not contain more than 15 elements.
5. A string may not contain more than 255 characters (including all asterisks but excluding the bracketing primes).

Appendix 2 Primitive functions

The following table contains the complete set of current primitive functions; this set is open ended and further functions can be (and have been) added quite easily as required. It should be emphasised that these functions are part of the translation scheme and do not constitute the user image.

Function	*Parameters*	*Action*
INTCH	$1, $2—integers $3—(optional) string	Return a string $2 characters long starting with $3 (if present), the remainder of the string being the least significant digits of $1 (no zero suppression).
GEN	$1—function of form @ output pointer# $2,$3, . . . strings	$2, /$3, . . . form an entry which is inserted in the output chain following the entry pointed to by VAL($1). The output pointer is updated to point to the last string inserted.
LOGOUT	$1—string	Output as specified by the output chain is generated and written to a local file specified by $1. If any failures have been detected, output is diverted to an error stream.
OPENOUT	$1—string	The previous text buffering file (if any) is closed and a new file, specified by $1, is opened.
COPYTO	$1—string	Lines are copied from the input stream to the current text buffer file until a line starting with $1 is found.
SWITCH	$1—string	Input is switched to a local file specified by $1. On termination of this file, input reverts to the current input stream.
IF	$1—integer $2, $3, . . .—functions	Evaluate and return $2, $3, . . . if $1 = 0, 1, ...
LOOP	$1—integer $2—function	Evaluate $2, $1 times, setting the 'previous statement' value * successfully to 1, 2, ...
VAL	$1—function or other value	Evaluate and return the result of $1 if it is a function—otherwise return $1 itself.
RETURN	$1—any	Exit from function, which returns $1 as its value. (RETURN is not required at the end of a function.)
TYPE	$1—any	Return an integer indicating the type of $1.
EQ	$1, $2—any	Return integer 0 (true) if $1 and $2 are equal in quantity and type—otherwise return 1 (false).
GT	$1, $2—integers	Return truth value of $1 > $2.
ADD	$1, $2, . . . integers	Return $1 + $2 + ...
RANDOM	$1—integer	Return random integer in the range 0 to ($1—1).
CONST	$1—any	Return $1 with 'constant' flag attached.
HOLD	$1—list	Copy $1 into permanent storage and return this copied list (an explicit form of the automatic facility described in the paper).
SET	(i) $1—function of form @identifier# $2—any	$2 is assigned to the identifier in $1, the assignment being not limited in scope.

	(ii) $1—list $2—integer $3—any	Element number $2 of $1 is assigned the value $3.
FAIL	$1—integer	A failure message indicating failure number $1 is generated.
LINECOUNT	$1—integer	Reset the line count, used in failure messages, to $1 and return the previous line count (used to exclude mapping functions from the line count for user failure messages.)
JOIN	$1, $2, . . . lists	Return the list formed by concatenating $1, $2, ...

Appendix 3 User documentation

The following extract from the mapping functions for the ICL Multijob system provides an example of the type of user information which can be provided from this source.

```
/++++++++++++++++OPTIONS - SETTINGS AND DEFAULTS++++++++++++++++++
/
/
/
/==========(1) - COMPILE OPTIONS
/
!LANGUAGE = FORTRAN;/                    SOURCE CODE LANGUAGE- ALSO :
                                           =ALGOL, COBOL, UCODE OR LSD
!CLTIME  =20;/                           TIME LIMIT FOR COMPILE & LINK
                                           IN ETU (1 ETU =3.5 SEC. CPU)
/
!CLSTORE =@IF(!MANYNAMES,200,128)\;/     STORE FOR COMPILE & LINK EDIT
                                           IN 512 BYTE UNITS; DEFAULT IS
/                                          128 FOR FTRAN4, OTHERWISE 200
/
!SOURCELIST =YES;/                       COMPILER SOURCE LISTING
!OBJECTLIST =NO;/                        COMPILER OBJECT CODE LISTING
!DEBUG   =NO;/                           COMPILER ETC. DIAGNOSTICS
!REFERENCES =NO;/                        =YES FOR COMPILER SYMBOL TABLES
/
/=====(1.1)- FORTRAN OPTIONS
MANYNAMES =NO;/                          =YES FOR LARGE TABLE COMPILER
/
/=====(1.2)- ALGOL OPTIONS
!ENTRY   =@$1 4\;/                       ENTRY NAME(DEFAULT IS FILENAME)
!ALGBUG  ='ROUTE,ASSIGN';/               ALGOL DEBUG FACILITIES
                                           (INVOKED IF DEBUG= YES )
/
/
/=====(1.3)- COBOL OPTIONS
!COBMAP  ='MAP,XREF';/                   COBOL REFERENCES SPECIFICATION
/
/=====(1.4)- USERCODE OPTIONS
!MACLIB  ='SYSTEM';/                     MACRO LIBRARY
/
/=====(1.5)- LSD OPTIONS
!GENTIME =50;/                           MACROGENERATION TIME LIMIT (ETU)
!GENSTORE=300;/                          MACROGEN. STORE LIM.(512 B)
!NEWIO   =YES;/                          - OPEN FN. USED FOR ALL STREAMS
                                           =NO FOR OLD I/O - PRINT &
/                                            READ OPENED AUTOMATICALLY
/
!LSDD    ='DECL';/                       LSD GLOBALS IN USER:GROUP.LSDD(S)
!LSDE    ='EXTS';/                       EXTPROCS IN USER:GROUP.LSDE(S)
!LSDSTACK=100;/                          SIZE OF LSD STACK
!MAIN    =YES;/                          =NO IF NOT MAIN MODULE
!LSDU    ='MICCSS';/                     USER NAME FOR LSD ROUTINES
!LSDG    ='NEWLSD';/                     GROUP FOR LSD ROUTINES
!LSDP    ='PROG';/                       LSD PROGRAM NAME
/
/
/========== (2) - LINK EDIT OPTIONS
/
!PROGMAP =YES;/                          PROGRAM MAP (SEE MAPLEVEL)
!MAPLEVEL ='MAP';/                       THIS GIVES MODULE MAP-
                                           ='XREF' ADDS CROSS REFS.
/                                          (NO EFFECT IF PROGMAP =NO)
/
!LET     =YES;/                          =NO FAILS LINK EDIT IF ANY
                                           UNSATISFIED REFERENCES
/
!ERREX   =NO;/                           =YES FOR ERROR EXIT FACILITY
!EXTO    ='DUMMY';/                      ERROR EXIT LABEL (LINKED TO
                                           UNSATISFIED REFERENCES)
/
!PROGSAVE =@!PROGFILE(NONAME)\;/         FILE TO HOLD LINKED PROGRAM
!LIBLIST =();/                           LIST OF SUBROUTINE LIBRARIES
/                                          TO BE USED IN LINK EDIT
```


References

BARRON, D. W., and JACKSON, I. R. (1972). The Evolution of Job Control Languages, *Software*, Vol. 2, pp. 143-164.

CALDERBANK, M., and CALDERBANK, V. J. (1973). A Portable Language for System Development, *Software*, Vol. 3, pp. 309-321.

HOPGOOD, F. R. A., and DAVENPORT, J. (1972). The Quadratic hash method when the table size is a power of 2, *The Computer Journal*, Vol. 15, pp. 314-315.

JACKSON, I. R. (1970). The Design and Implementation of Command Languages for Digital Computers, Ph.D. dissertation, University of Cambridge (reproduced as NTIS no. DB 197307).

MCGREGOR, D. R. (1972). The Culham Job Control Language, SIN 2/72, UKAEA Culham Laboratory, Abingdon, Berks.

RICHARDS, M. (1969). BCPL: A Tool for Compiler Writing and System Programming, *SJCC*, pp. 557-566.

STOY, J. E., and STRACHEY, C. (1972). OS 6—An experimental Operating System for a small computer. Part I: General Principles and Structure, *The Computer Journal*, Vol. 15, pp. 117-124.

WAITE, W. M. (1970). The Mobile Programming System: STAGE 2, *CACM*, Vol. 13, pp. 415-421.