COMMAND LANGUAGES AS THE USER'S INTERFACE TO THE COMPUTER SYSTEM

F. Hertweck

Max-Planck-Institut für Plasmaphysik, Munich, Germany

Abstract

Most users of scientific computer systems are familiar with FORTRAN, but they experience difficulties when they actually want to execute a program: it has to be embedded into a sequence of "job control statements" before the operating system would accept the program. There seem to be as many job control languages as there are computing systems.

This incompatibility between systems, the poor design of many of these "languages", the lack of program and data portability has recently aroused new interest in the subject.

This lecture will give the student an introduction to the subject. After a review of the present state of the art the general considerations and the basic concepts relevant to the design of a CL are discussed. The CL will be looked upon as the interface between the user and the computing system. This interface becomes especially important if we think of the computing system as a network of many different computers. Then more specific topics will be discussed: The access to and handling of data files, the execution of programs and the embedding of programs into the CL environment.

1. Introduction

1.1 Command Languages as User/System Interface

When a "user" wants to solve a problem with a computer, he usually will proceed in two steps: He will design an algorithm which will solve his problem and construct a program which is supposed to represent that algorithm, and then he will attempt at running that program on a computer at hand. While for the first step he may use one of the standard programming languages, like FORTRAN or ALGOL, which are more or less the same on different computers, the "approach to the computer" will depend very much on the type of computer and operating system. Experience tells us that the average user, normally not interested in the intricacies of the "job control language", faces considerable problems. The reason is that this interface to the computing system (including both hardware and the operating system) is rarely well defined (or, rather, defined in some ad hoc manner). In this lecture we will discuss the general considerations relevant to the design of a Command Language and the basic concepts upon which a CL should be built. The CL will be considered to be the universal interface between the user and the computing system. Later on we will discuss some more specific topics like the access to and the handling of data files (to-day still a major source of incompatibilities between computing systems), the execution of programs (i.e. the description of programs in a way convenient for the user) and the embedding of programs into the CL environment.

1.2 A Survey of the Past Development

Command Languages have developed along with the development of computers and their operating systems. Until to-day, not much systematic research has been devoted to this subject but most of the development has been left to the manufacturers of computing systems.

In the early fifties, computers were operated by direct man/machine interaction. No operating system and no CL was needed. As systems grew bigger and faster, this approach turned out to be too uneconomical to be sustained. Therefore the first "control card systems" were designed. Programs from several users, with some suitable "magic cards" put onto the front and the end of a program, were composed into a batch of programs to be processed sequentially. A typical example is the IBSYS system for the IBM 7o9o computers.

As the technology of computers advanced further, the multiprogramming systems were developed. Still to-day, together with the interactive (or time-sharing) and minicomputer systems, they play an important rôle as the general "computing center" machines. For those batch oriented systems with yet considerable file handling capabilities more advanced versions of the control card systems were developed. As an example let us mention IBM's JCL/36O, remarkable for both its richness of function and unwieldiness to use.

But some other examples exist too, more up to the state of the art when compared with programming languages: Burroughs Work Flow Language and, even more noteworthy, ICL's control language SCL. Typically, they are both featuring some of the structure of Algol.

That the development up to now is insufficient becomes apparent if we want to address interactive computing systems. While batch systems could be adequately handled with the static description by JCL statements, interactive systems require a continuous dialogue of commands (by the user) and responses (by the system).

1.3 Standardization Activities

During the last five years or so several attempts have been made to assess the existing CLs and to investigate the possibilities for standardization. It became apparent that there was such a need for standardization, as much as for the programming languages. We do not want to go into the details of those activities but a few remarks seem appropriate [1].

The ANSI Operating System Control

Languages Study Group is the earliest of those activities. In 1971 the following recommendations were presented: The need for a standard is pressing and its attainment should be possible; none of the OSCLs surveyed was considered a suitable candidate for the standard; there should be only one standard OSCL.

The Dutch JCL committee developed a list of basic job control functions: allocation of resources, conditional selection of (part of) a job, execution of a program, declaration of job attributes, synchronization of concurrent computer programs.

The CODASYL (Conference on Data Systems Languages) OSCL Task group put its emphasis on language production rather than on the issues of standardization: Investigation of the functional requirements for communications between users, "functional programs" and hardware (i.e. of the OSCL); determination of functions necessary to define a standard OSCL interface; development of linguistic elements which drive these functions; definition of a machine-independent OSCL.

Other groups to study the OSCLs have been formed in various professional societies, like IEEE, ACM and BCS (British Computer Society). Also specific computer user groups, like SHARE or the ARPANET users have discussed the problems, and recently IFIP has established a Working Group to study operating system interfaces.

2. Present State of the Art

2.1 Review of Recent Literature

In the recent years several papers have been published on the subject of Command Languages. We will mention a few that taken together give a fair representation of the views and opinions of the computer community.

Barron and Jackson compare the two command languages for the operating systems OS/360 and GEORGE 3.[2] Introductory to the comparison is a review of the historical development of those two languages. The authors point out that JCL/360 closely resembles assembly language, whereas GEROGE 3 command language looks more like a simple high level language. Finally, the issue of defining a command language as an extension to an ordinary programming language, and some more recent developments are discussed.

The paper of Dolotta and Irvine reflects the dissatisfaction of users with the existing command languages, especially for interactive work.[3] Therefore a system is demanded with a machine-user interface that can be tailored by each user to fit his own needs, and which is simple and extensible. It is recognized that this imposes certain requirements on the structure of the operating system.

Stephenson takes a similar position: a CL is by definition interactive.[4] Commands should be issued both from within programs as well as from outside a program. The user may create certain "Command Environments" which may be considered as dynamically defined scopes for commands.

The MULTICS System was the first serious attempt to create an interactive "computer utility".[5] Specially noteworthy are the symbolic segmentation of programs and data, and the file system hierarchy, with its naming conventions and access control emphasizing sharing of information between users.

In most operating systems the command language is separated from the programming languages used. Lauesen and Jensen take the opposite point of view: It is proposed to extend existing programming languages with a set of operating system control facilities. This would make special command languages superfluous and would allow even the writing of operating system modules in a higher level language.[6][7]

Parsons has taken a high-level language approach[8] and defined a powerful, flexible and machine-independent control language, in which many of the desirable facilities that are available in modern programming languages have been included.

A recent very interesting paper by Brunt and Tuffs describes the approach adopted for SCL, the control language of ICL's new 29oo series of computers.[9] The design emphasis of SCL is "Usability", in view of the fact that a large diversity of users require it.

2.2 Command Invocation and Command Environments

Some systems are batch oriented, like OS/360.[10] This fact is reflected in the form of the JCL: It can practically only be used for the invocation of programs ("jobs"). There are basically three commands: the JOB, the EXEC (=execute program), and the DD (=data definition) statements. While the EXEC command is similar to a statement in a programming language sense, the JOB and DD statements more have the character of declarations. It is typical for such a command language that it is compiled (by the "Reader/Interpreter") into control tables. In TSO (the "time sharing option" of OS/360) the concepts of JCL/360 are extended to an interactive system.

Stephenson, as already mentioned, takes a different view: CLs should be interactive. In GEORGE 3 the approach is more comprehensive, considering interactive mode and batch mode as two natural ways to use CLs. This seems to be a widely recognized requirement.

There is another observation one can make about JCL/360. The DD "statement", basically a declaration, nevertheless also performs to some extent actions like a command: it can be used to catalogue or delete files that have an existence outside the job. Sometimes only those side-effects are of interest and so the user is forced to execute a trivial program (consisting of a return to the operating system only) to carry those DD statements.

In some of the systems, a command cannot be distinguished from a program. The "command verb" is in fact the name of a file which is considered to be an executable program. Obviously, in such a case naming difficulties may arise - either that the name of the file is not suggestive as a command verb, or that for an existing command verb another file (a new version) is to be selected. The latter case can be handled by suitable scope rules.

The form of a command is typically

⟨label⟩ ⟨verb⟩ ⟨parameter list⟩

where the label is optional; the structuring is that of an assembly language statement or of a high level language procedure call.

Frequently, this is the only form; it makes any structured programming impossible. The main difference to the programming language procedure calls is the possibility of using keyword parameters (of the form ⟨keyword⟩ ⟨value⟩ or ⟨keyword⟩=⟨value⟩), which obviates the requirement of strict order or completeness of the parameter list.

Many systems or proposals require commands to be issued from within programs as well as from outside a program. Surprisingly enough, the consequences of such a possibility are not much discussed: if the "command" is just another program, there is not much difference to a subroutine called; if, however, the command may unexpectedly ask for resource allocation, the danger of deadlock is immanent if no overall resource scheduling is stipulated.

Finally, we should mention the problem of scope of command definition. Some authors ask for scope rules. They are called context, or environment, or level and mean, in fact, the same: by issuing certain commands, the command interpreter may perform a transition into another state, where other commands are defined than before. The term "level" suggests a nested structure, whereas "environment" is more general. Though such a facility is claimed to be desirable, the possible consequences for the user seem to be neglected; he may have difficulties in keeping track of the sequence of contexts he is using.

2.3 Invocation of Programs

A command language is not a primary tool, like a programming language which is used to write problem-solving algorithms, but rather a secondary tool to execute programs conveniently. Therefore, the invocation of programs of jobs assumes a prominent place. As already mentioned, it is the only purpose of JCL/360 where a job is defined as a sequence of job steps, each requesting the execution of a certain program given as a load module. Such a job is considered independent of any other jobs. (The only interaction with other jobs is indirect, by competing for common resources). In other systems, like GEORGE 3 or BOSS 2, a job is defined as a sequence of commands, obeyed one after the other. If a job is an offline job, the sequence of commands is taken from a file, if it is an online job, the commands are read one by one from the console, thereby giving the user opportunity for interaction with the job.

One may view OS/360 as a very primitive "virtual machine", capable only of executing programs defined as (relocatable) core images. For the user, this results in the tiresome host of job steps he has to deal with. On the other hand, one could imagine an operating system, which would consist of a set of virtual machines, capable of executing FORTRAN, ALGOL, PL/I, etc. programs. Attempts to implement such virtual machines are widely known as "fast batch monitors" (like, for instance, WATFOR). Another example is the APL machine or the BASIC machine.

A program may be viewed as a complete algorithm to perform a certain data processing task. What normally remains undefined until program execution is the input source and the output destination; some suitable declarative commands define them when the program is invoked (DD statement in JCL/360, ASSIGN statement in GEORGE 3, etc.).

2.4 Procedures and Macros

Before discussing the use of procedures and macro facilities in command languages we will review their use in ordinary programming languages.

Procedures are constructs found in all higher level languages with different degrees of complexity. Algol (or Algol W) would be one of the more advanced examples: procedures may be recursive, parameters may be defined as input values, or result parameters, or they may be "called by name". In FORTRAN, parameters are called "by reference". Also, procedures may be passed as parameters to other procedures. The programmer need not know anything about the internal structure of the procedure (except if it refers to global entities). In short, a procedure is defined by its actions in terms of the returned results on given input parameters.

Macros are used mainly in assembly languages for two purposes:

- to expand "short-hand" code into standard in-line code patterns (used as a substitute for procedure calls) - thus economizing program writing locally.

- to produce different versions of an algorithm by conditionally generating different sequences of code dependent on compile-time parameters - thus economizing program writing globally.

In this context we are mainly interested in the first type of macro usage. Typically, macros assume the form of procedures with strings as actual parameters. These strings are inserted wherever the corresponding formal parameters appear in the macro body.

It is probably due to the fact that existing command languages have been developed in analogy to assembly languages that they frequently lack procedure facilities and have some macro facilities instead. On the other hand, this is consistent with the lack of well-defined data types and corresponding variables, so that only the primitive string substitution techniques are usable. Also the DD-statement replacement or modification technique in JCL/360 is an example of this kind.

2.5 Files

Practically all CL systems assume the existence of a filing system. In fact, it would be impossible to have an advanced interactive system without files. However, as far as the implementation is concerned, there are as many different approaches as there are different systems.

In OS/360, files (at least when manipulated in the command language) are considered very much under the aspect of their physical attributes, like block sizes, allocation quanta, devices, etc. If a file is not explicitly catalogued, it cannot be accessed without indication of the storage volume it resides on. There is no doubt that under certain circumstances the physical attributes must be accessible, but it is objectionable to build a system such that these attributes have always to be specified.

The other extreme is probably MULTICS, where files are synonyms for segments (=address spaces), i.e. files are considered to contain directly accessible data. MULTICS has a hierarchical structure in the filing system, such that a file is identified by a "path name" in the hierarchy. Thus the constituents of that hierarchy are file directories and the files themselves. File directories contain entries pointing to files or to other directories. Each file or each directory may have access attributes for each user.

We will return to the subject of file access later.

2.6 Command Languages viewed as Programming Languages

Considering the development of more and more complex command languages and the complicated sequences of job steps that users may - and as a matter of fact also do - compose, it is natural to consider the command language as a type of programming language working in a special environment, the virtual machine, consisting of the computer hardware equipped with the system software. Viewed in this light most of the existing command languages are utterly primitive and awkward: variables are not available at all or - at the best - only as "return codes" or some other severely limited data type; flow control consists of a simple "local" conditional execution of one job step (IF...THEN execute next job step, OTHERWISE skip it) and a GOTO (forward only) - or it consists of nothing at all; procedures or macros most often are available only with restrictions of some sort (nesting depth, or parameter forms, or the like); the syntax of most command languages is primitive, inconsistent and difficult to remember for the user.

Most CLs resemble an assembly language. But there are two more advanced examples: Burroughs Work Flow Language (WFL) and ICLs System Control Language (SCL). Both have an algol-like structure.

2.7 Command Language built into Programming Language

Viewing command language as a high level programming language, just with some special facilities, leads to the idea of building command language into one or all of the existing programming languages;or rather to extend the programming languages with the necessary job control functions. The programming languages already contain several types of data and variables, elaborate flow control and procedure facilities, and in certain programming languages (PL/I, Cobol, RC 4000 Algol, to mention a few) some job control functions are available.

In order to make a normal programming language suitable as a CL, it should be extended by: declarations for new data types like files, devices, storage, CPUs (and with attributes regulating the access rights to these resources), jobs (and other processes) and message buffers; the definition of a standard set of procedures to invoke the most frequently used CL functions.

With these facilities available in all the languages of a certain system, the only "external" control command is the logon command where the user states his identification and the programming language he wants to use.

Arguments for incorporating the command language into the programming languages are:

- it is simple for the simple-minded user who needs only a few of the standard procedures and may think and plan completely in his own programming language.
- It is easier to learn, being one language only for the user, and it allows him easier control over the operating system.
- It makes the ordinary programming language useful over a wider range of applications, including writing of operating systems and pseudo-drivers.
- It allows the programmer to get nearer the kernel of the operating system when the programming language is used as implementation language for the higher level layers of the operating system.
- It does not add to the proliferation of computer languages.

Arguments for keeping command language and programming language separate run as follows:

- It is simple for the simple-minded

user, because he needs only very few "magic words" of the command language and never thinks of it as a new language.

- Programming language and command language work in rather different environments. The first one should be problem oriented, the last one system oriented. Therefore they most probably don't coincide even if they do have certain elements in common.

- The rules of structured programming teach us to keep the different levels of a problem separate. Since the job control and the internal algorithm of a program are different levels we should help all users to keep them separate.

- The new data types are related to the environment of the job and must therefore follow other scope rules inconsistent with the existing data types in the programming languages, where the scope is related to the flow of the algorithm, independent of other processes.

- The compilers for command languages and programming languages should emphasize different aspects: in programming languages the effective handling of simple data structures, in command languages the effective handling of files, resources and unusual system conditions.

- The economics are against the integration: Implementation of all job control features in all programming languages is much more expensive than implementing a standardized, separate command language. (Here is foreseen a continued proliferation of more or less special purpose programming languages).

I feel that all in all the latter arguments carry greater weight than the former ones. But it may be added that still the most important problem is to find the really fundamental data types and operators for the job control algorithms.

2.8 Other Considerations

We shall add a few remarks about some other aspects of a CL. An important consideration is its simplicity. Since all users must use it, and since it does not directly contribute to problem solving, the user should be relieved of any undue burden. A clean and consistent syntax and as few well defined command levels as possible would contribute towards this goal.

A CL should also be extensible in order to accommodate new hardware and software. We may expect that a computing system will grow and that it will be required to provide new functions. It should be applicable both in interactive and batch applications, be indifferent against the various programming languages and have a high degree of portability. Default and error handling should be convenient and foolproof.

3. General Design Considerations

So far we have mainly presented a critical review of existing command languages and some ideas for their development. We will now summarize what we have learnt into some general considerations that seem to be important for the design of a command language. They may be grouped in four categories: aspects of the user/system interface, aspects of CL structure, aspects of CL objects, and aspects of user mangement.[11)]

3.1 User/System Interface Considerations

The command language is the universal interface between user and system. The term "user" has to be understood in a very broad sense: System management staff, systems programmers, operators, application programmers, users of program packages; in fact the whole span between the occasional user and the professional.

The command language is primarily interactive, terminal oriented. However, the specifications and the design of the language should be independent of whether the supporting system offers features like dynamic relocation or paging. Moreover, the same command language must be usable in a batch environment. There should be no distinction on whether a job is invoked from a terminal or entered from the central card reader, or from a remote job entry terminal. (This includes, for instance, a geographically invariant identification of the standard line printer).

The command language must cover the whole span from the most simple usage of the system by the naive user, to the most sophisticated applications (like, for instance, tuning the system for better performance). In other words: simple things should be expressible in simple ways, detail should be added only when necessary.

Considerable attention has to be paid to the foolproofness of the command language. "Dangerous" commands, like deletion of a file, etc., have to be reconfirmed by the user when using the system interactively. Any error, syntactical or other, switches the user process (="terminal session") into "correction mode", so that the command may be immediately corrected.

The command language must be constructed in such a way that it requires only a minimum of changes to existing languages or their compilers. However, all executing programs should talk to the system through a standard interface; a running program must be free of anything like "I/O-packages" or "interface routines". In other words, the system interface must be standard, comprehensive and complete.

A comprehensive and versatile text editor should be part of the command language environment. This reflects the fact that creation and modification of source data is the user's main activity.

A certain set of standard utility programs, frequently needed by almost every

user, should be part of the CL. Examples are to copy a magnetic tape or to punch out a deck of cards.

3.2 Structural aspects of Command Languages

The command language must be well structured, with a syntax that allows automatic parser generation for both compilation and interpretive execution of command language programs.

In the same way as a programming language is based on the duality between data and operations, a command language is based on the more general concepts of objects and actions; for instance, the action "switch line to modem" operates on the objects "telephone line" and "modem". In both cases some control structures are needed in addition in order to execute operations/ actions conditionally.

There is no need to introduce some awkward control structures, because there are programming languages with excellent features of this kind (Algol W, Simula, PL/I, Pascal). It seems appropriate to select one of those languages and to take over its control structure. Moreover, one could visualize several semantically identical versions of the command language, based on different or resembling different programming languages.

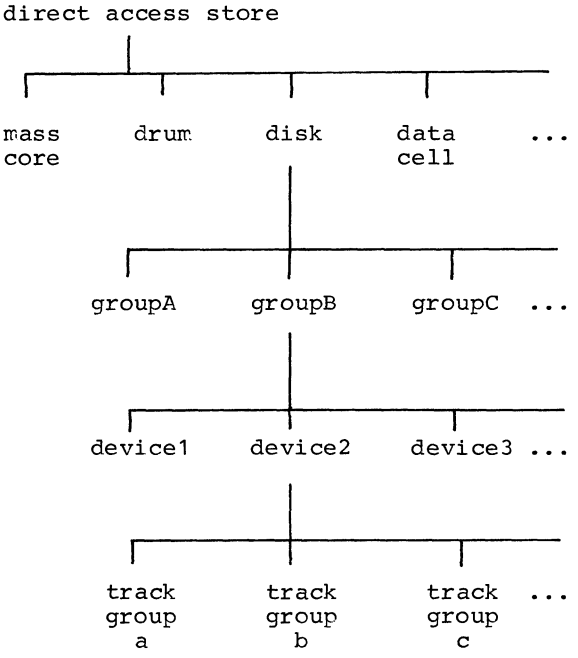
It should be unnecessary to mention that a command language must have a very elaborate procedure facility. Computations, defined by programs written in other languages should be treated as subroutines at the command language level. If the command language permits specification of parallel execution of statements (which it should!), the concurrent execution of several programs, even written in different languages, is almost trivial.

The command language must easily extensible by addition of new entities belonging to predefined classes, or by adding new syntactical constructs. The first kind of extensibility requires that certain syntactical units (like data types in Pascal) be defineable; this type of extension would be preplanned and built into the language. The other kind of extensibility implies that the syntax analyzer and parser are planned for easy extension.

A programming language normally defines all of its variables internally; the variables have no existence outside the program. But there are exceptions: there exist some "predefined variables", or "logical file names", etc., nonlocal to the program. In a command language the existence of such objects is the rule: there are named objects that exist outside a command language program (though they are created within such programs!). An important point in the design of a command language is the honest establishment of means to create and access those permanent, external objects.

A command language cannot fulfill its purpose, if its objects are just complementary to those of the programming languages; a certain common subset of data structures must exist, for instance, in order to be able to pass parameters to a program. This calls for a variety of data types (variables) of the usual kind to be defined in both the command and programming languages. (As a consequence, the passing of information between sequential or concurrent programs becomes quite easy).

From the considerations mentioned in the beginning it follows that a kind of hierarchical naming system must exist (similar to the MULTICS file names) for all kinds of objects. An example would be the hierarchy:



For each node, access rights must be specified for each user (for instance, only systems programmers would get down to the level "device" or even "track group", the normal user possibly only knowing of "direct access store", as compared to "sequential stores"). At the same time, this naming hierarchy with access right specification establishes the possibility of controlled sharing of all kinds of objects.

Contrary to a programming language, where practically all data objects are declared explicitly, in a command language many objects are permanent objects, existing independently of a command language program. Therefore special attention must be paid to scope rules. It is recommended to adopt the well-established concept of the Algol block structure. In addition there should be a facility to screen objects and actions from outer blocks by affirmative or negative enumeration ("none, except ..." or "all, except ...", respectively). Scopes of objects and actions must be part of the naming hierarchy. Since the

block structure is suitable only for automatic creation and deletion of objects, other facilities must exist to create or delete permanent objects.

3.4 Aspects of "user management"

From 3.1 follows the need of a comprehensive "user management system" to establish users of the system with their account information, access rights, resource grants, functional scope, etc.

A central facility is needed, which may be called a "logfile", associated with each user. It is used for recording all activity of a user. All system components know of the existence of the logfile and will place messages into it when appropriate. In a hierarchical user group environment (like tutor/students, work team leader/members) the group head may have access to the logfiles of the members (even with higher access privilege than the members). The logfile must be structured such that parts may be retrieved, copied or extracted (for instance, to save an updated table of contents of a storage volume). It also should be possible, subject to access rights, to send messages to other user's logfiles.

A "user process" is created by "logon". At the same time, an initial environment (scope) is defined. If the user belongs to a group of users, the initial scope may be defined by the user on the next higher level (the group head). A user process is terminated by "logoff".

4. The Basic Concepts of Command Languages

When looking at the multitude of keywords and special concepts in the CLs of today, it becomes clear that both a unification and an "orthogonalization" of concepts are needed. By the latter we mean that different concepts should be easily combined.

In this section we will attempt to present the basic concepts of a general command language, keeping in mind that it must handle a very general class of objects and associated actions.

4.1 User Processes

By "user process" we denote a process which is associated with a user. It will be created by the system manager and will have some attributes like: the user-id, the initial password, account information, resource grants, the privilege level, etc. This user process exists as long as the user has permission to use the system.

The user may activate by a "log-on" command this process. At the same time he may specify resource claims, account information and the initial scope of commands. It is this process with which the user "talks" by entering commands and accepting the responses. A "job card" would be the batch version of the log-on command.

The "log-off" command will make the user process dormant again. At that time the user may specify disposition information for objects he has created or used like, for instance, the processing of the log-file.

4.2 Objects and their attributes

The objects of a CL fall into three classes:

(1) Device-like objects, like: working store, CPUs, peripheral devices, data channels, communication lines, storage volumes, etc.

(2) Variable-like objects. They may be local variables, private to one process and declared in the CL program. Data types would be logical, integer, real, character, reference. The variables may be simple, arrays or records, or files. Then there may be shared variables, accessible to two or more processes. Some variables may be global and accessible to all processes (for instance: the user directory, time and date, etc.). Other variables, like files, semaphores, message queues, etc. are shared only between few processes.

(3) Processes: There are system processes, like the spool processes, I/O drivers, etc. Then there are the system operator process and the user processes.

To almost all of the mentioned objects attributes can be attached. They may be defined statically or dynamically.

(1) File types. It seems reasonable to classify files according to their general kind of contents. We may distinguish among text files,executable programs, data files, CL files, directories, etc.

(2) Access rights. An object of the CL may be subject to access restrictions, like being private, shared, or public. A file may be accessible to a user only for read operations or execution, or he may extend, modify or even rewrite it.

(3) For objects a scope may be established which again controls lifetime of and access to those objects. Some objects may be confined to blocks (like in some programming languages), others may already exist, but a subset of them is excluded by a specification like "all, except..." or "none, except...". Furthermore, objects may be temporary or permanent.

(4) In many instances maximum resource requirements have to be given, like maximum execution time for processes, maximum size for storage and files, maximum number of accesses to an object, etc.

(5) Other attributes refer to the logical structure of objects, especially files. Specifications like record structure, access modes, etc. belong to this kind of attributes.

(6) In some cases information about physical attributes of objects has to be supplied. This should be confined as far as possible to the purpose of system tuning,

etc. Device types, device addresses, blocking factors and similar information belongs to this class.

4.3 Declarations

As in programming languages, a CL will have the facilities to declare objects. Obviously, in a CL they will be to some extent different in nature.

(1) Procedure declarations in a CL will contain declarations of objects local to that procedure and actions. An important feature would be a general keyword scheme for the specification of actual/formal parameter correspondence. It seems reasonable to permit procedures written in other languages too. This provides a consistent way to "execute programs".

(2) Object-process links are declarations that resemble the actual/formal parameter correspondence, except that there are usually more attributes that need to be specified (usage modes, access right restrictions, etc.).

(3) When new objects have to be created, this frequently can be done by a declaration. These declarations must permit the specification of attributes, initialization and scope rules.

(4) To replace the macro facility one may introduce "structure declarations" which define frequently used parameter lists, attribute lists, etc.

4.4 Actions

Actions in a programming language are usually called statements and are mostly confined to computing new values for variables. In a way, one can consider I/O statements as exceptions to that rule. Since the objects in a CL are of much wider scope, so are the actions that deal with them. We may classify actions into three groups:

(1) Manipulation of objects: Arithmetic and logical expressions perform calculations on local and shared variables. Concatenation and decomposition operations may be used to handle files. The assignment statement may be viewed in a broader scope: one may assign new values to local variables, new records to files, or volumes to devices. Also the synchronizing operations "signal" or "send message" may be considered as assignments to shared variables of type semaphore or message queue.

(2) Creation/deletion of objects is another kind of action needed for those objects for which there are no well defined scope rules (for example, creation/deletion of permanent files). A corresponding example in programming languages would be the generation of record-variables. Another kind of action belonging to this class is the (dynamic) changing of attributes of objects.

(3) Execution of programs may be performed in one of three ways:"Normal execution" is the interactive (or semi-interactive) execution of a program as part of the user process. The user may either conduct a dialogue with the program or he may observe its progress by inspecting logging messages at his terminal. "Independent execution" is the usual batch execution mode, where execution of a program goes on independently of what the user does at his terminal. "Parallel execution" is the active cooperation of two or more processes.

(4) Similar to the facilities in programming languages one needs control statements in the CL. We have already mentioned CL procedures; then one would need the if-then-else, the case-statement, some loop control like while-do or repeat-until. It seems reasonable to have facilities for handling exceptions (like the ON-statement in PL/1) and the synchronization operations "wait" and "await message" belong to this class of commands too.

4.5 Summary

Though the ideas and contemplations, presented in the last two sections, are to be considered preliminary, nevertheless it seems that the present concepts and tools used for the definition and design of programming languages are also well suited for the definition of the more general command languages. A basic requirement, however, is to define the structure of an information processing system in as abstract terms as possible, without neglecting the fact that the underlying hardware may sometimes show some peculiarities (as a result of the attempt to build the data processing system in an economic way). It seems important to incorporate the "user", who is continuously interacting with a system, into that system as an integral part; this is reflected in the emphasis given to "user management" and "user process". The term "information processing system", as used above, is to be understood as the union of the data processing system and the set of users.

5. File Access

In most programming languages the concept of file is implemented. In some it may be a direct language construct, like in PASCAL or PL/1, or it may be defined indirectly by statements in the language, like in FORTRAN, or it may be implemented by a set of (standard) procedures, like in ALGOL 6o or ALGOL W. In all cases it is assumed that files exist as "data sets" on an external storage medium, though this is not so obvious in the case of PASCAL.

While many simple statements, like assignment statements or go-to statements look almost identical in all the mentioned languages, there is quite a bit of difference in other statements (like for statements/do-loops), and the situation is worst in the area of files and I/O statements.

This survey tries to bring some order into the different views. First, as the most typical examples, PASCAL, FORTRAN and PL/1 are examined regarding their I/O facilities. Then a general approach to the description of sequential and indexed files is attempted. It reflects the view that a program should not be concerned with the physical properties of a file, but only with its logical properties. In other words, files are exclusively viewed as data structures. Any physical properties of files should be defined outside the program - for instance when associating a (physical) data set with a (logical) file in the CL environment of a program. But even at that stage the specification of physical properties may be superfluous if the file already exists. It is assumed that it carries with it its physical attributes (stored away at some convenient place).

5.1 PASCAL Files

A PASCAL[12] file is a sequence of components (data types) of the same type. A natural ordering is defined through the sequence. At any instant, only one component is directly accessible. Other components are made accessible by progressing sequentially through the file. A file is generated by sequentially appending components at its end. Therefore, the file type definition does not determine the number of components. A file with 0 components is called empty, the number of components is called the length of the file.

Every declaration of a file variable f with components of type T implies the additional declaration of a "buffer variable" of type T. This buffer variable is denoted by f↑ and serves to append components to the file during generation, and to access the file during inspection.

A standard file type is the textfile, which is a file of characters. The standard file variables input and output are predeclared as textfiles. A PASCAL program should be regarded as a procedure with these two variables as formal parameters. The corresponding actual parameters are expected to be specified by the system command activating the PASCAL system.

At any time, only the one component determined by the current file position is directly accessible. This component is called the current file component and is represented by the file's buffer variable, denoted by ⟨file variable⟩↑

The following standard procedures are defined:

put(f) appends the value of the buffer variable f↑ to the file f. The effect is defined only if prior to execution the predicate eof(f) is true (i.e. the file is positioned at its end). Then eof(f) remains true, and f↑ becomes undefined.

get(f) advances the current file position to the next component, and assigns the value of this component to the buffer variable f↑. If no next component exists, then eof(f) becomes true, and the value of f↑ is not defined. The effect of get(f) is defined only if eof(f) = false prior to its execution.

reset(f) resets the current file position to its beginning and assigns to the buffer variable f↑ the value of the first element of f. eof(f) becomes false, it f is not empty; otherwise f↑ is not defined, and eof(f) remains true.

rewrite(f) discards the current value of f such that a new file may be generated. eof(f) becomes true.

eof(f) is a boolean function which indicates whether file f is in the end-of-file status.

Two procedures are used to handle the standard textfiles input and output (ch denotes a variable or expression of type char):

read(ch) means: begin ch := input↑; get(input) end

write(ch)means: begin output↑ := ch; put(output) end

In addition to these procedures, an actual PASCAL implementation (on the CDC 6400/6600) allows the use of a specifier within a file declaration:

IN the file is an external file (e.g. a permanent file connected to the job through a SCOPE ATTACH command). This file is to be read only.

EXT as IN. However, the file may be extended by appending further components at its end.

OUT the file is not discarded at the end of the PASCAL run, but remains available (e.g. to be made permanent by a SCOPE CATALOG command).

PRINT the file is printed after termination of the job.

PUNCH the file is punched after termination of the job.

In all cases, the first seven characters of the file variable identifier are used as its logical file name (as defined by SCOPE).

5.2 FORTRAN Sequential Input/Output

In FORTRAN IV[13] there are five sequential I/O statements: READ, WRITE, ENDFILE, REWIND, BACKSPACE. Their action is defined as follows:

READ (dsrefn, fmt, END = label1, ERR = label2) list

where:

dsrefn is a data set reference number (integer)

fmt is optional and either the label (=statement number) or the array

name of a format statement, or a namelist.

label1 is the label to which transfer should be made if an eof condition is encountered.

label2 is the label to which transfer is made if an error has been detected during data transfer.

list is an I/O list, i.e. a list of variables or array parts.

Note that END=label1, ERR=label2, and list are optional. The order of END= and ERR= may be reversed.

Transfer is made to the statement specified by the END= parameter when the end of the data set is encountered; i.e. if a READ is executed after the last record of the data set has already been read. (No indication is given of the number of list items read into before the end of the data set was encountered). If the END parameter is omitted, program execution is terminated upon encountering the end of the data set.

Transfer is made to the statement specified by the ERR= parameter, if an input/output device error occurs. No data is read into the list items and no indication is given of which record or records could not be read, only that an error occured during transmission of data. If the ERR parameter is omitted, program execution is terminated when an input/output device error occurs.

In the form "READ (dsrefn) list", a single record from the data set associated with data set reference number dsrefn is read and the values are assigned to the variables given in the list. This statement is used to read records written by a "WRITE (dsrefn) list" statement. If the list is omitted, a record is passed without being processed.

WRITE (dsrefn, fmt) list
where the parameters are as for the READ statement.

The form "WRITE (dsrefn) list" is used to write a single record from the variables whose names are given in the list into the data set associated with data set reference number dsrefn. The list cannot be omitted.

ENDFILE
where dsrefn is a data set reference number. The END FILE statement defines the end of the data set associated with data set reference number dsrefn.

REWIND dsrefn
where dsrefn is a data set reference number. The REWIND statement causes a subsequent READ or WRITE statement referring to the data set with this dsrefn to read data from or write data into the first record of the data set.

BACKSPACE dsrefn
where dsrefn is a data set reference number. The BACKSPACE statement causes the data set associated with dsrefn to backspace one record. If the data set associated with dsrefn is already at its beginning, execution of this statement has no effect.

5.3 PL/1 Input/Output

Since PL/1 I/O is very sophisticated, we will only present a very short survey of its main features. For details, see reference (14).

5.3.1 Stream and record oriented transmission. A collection of data external to a program is called a data set. The program only knows of the corresponding symbolic representation of data, called a file. PL/1 I/O statements are only concerned with files which can be associated with different data sets at different times during execution of a program.

There are two basic types of data transmission:

(1) stream-oriented transmission, where the organization ot the data in the data set is ignored within the program, and the data are treated as though they were a continuous stream of characters; data are converted from internal form to character form on output, and vice versa on input. Stream I/O is therefore ideally suited for card input and line printer output.

(2) record-oriented transmission, where the data set is considered to be a collection of discrete records. No data conversion takes place during transmission. Record oriented I/O is more versatile with regard to processing modes and types of data sets that can be handled. Since the transmission is a one to one mapping, any data format is acceptable for record I/O. The programmer must have knowledge of his data (record) structure.

5.3.2 Data sets. The following characteristics of a data set are recognized: Data sets are located on volumes, which are physical units of data storage (disk pack, magnetic tape reel). Data items within a data set are arranged in physical groupings called blocks (or, in teleprocessing, messages). A block may consist of one or more logical subdivisions, called records. (Sometimes, the term physical record is used as synonym for block, and the term logical record as synonym for records). When a block contains more than one record, the records are said to be blocked. Input/output statements of PL/1 generally refer to records, not to blocks.

5.3.3 Files. They are symbolic representations of data sets within a program. The notion of file stresses the logical aspects of a data set by determining how input/output statements access and process the associated data set. PL/1 requires a file name to be declared for each file, and it allows certain characteristics of a file to be described by file attributes.

The first group are the alternative

attributes:

STREAM/RECORD: STREAM causes a file to be treated as a continuous stream of data items code in character form, RECORD causes a file to be treated as a sequence of records, each consisting of one or more data items recorded in internal form.

INPUT/OUTPUT/UPDATE: INPUT and OUTPUT mean that the file is to be read or written only, respectively. With UPDATE specified the file will be used for both input and output and records may be inserted, deleted or altered.

SEQUENTIAL/DIRECT/TRANSIENT (with RECORD only): Successive records of the file are accessed on the basis of their ordering, if SEQUENTIAL is specified. With DIRECT the record is accessed by specifying a key which is assumed to uniquely specify the position of the record in the file. TRANSIENT is used for telecommunications only.

BUFFERED/UNBUFFERED (only for SEQUENTIAL or TRANSIENT): Records are transmitted through a buffer with the size of a block thus permitting I/O transmission concurrently with processing, if BUFFERED is specified. Otherwise, with UNBUFFERED specified, I/O organization is the responsibility of the programmer.

The second group of attributes are called additive because they may be specified in addition to the others. PRINT (for STREAM OUTPUT) indicates that the file is eventually to be printed. Thus the first byte of each sequence of characters describing a print line is reserved for printer control. BACKWARDS (used with SEQUENTIAL RFCORD INPUT files on magnetic type) specifies reading the records of the file in reverse order. KEYED specifies that a key is going to be used (for DIRECT), and EXCLUSIVE will lock a record for a RECORD DIRECT UPDATE file when accessed by a task.

5.3.4 Physical organization of data sets. Though it is claimed that PL/1 treats I/O on a logical level (i.e. by the access of the symbolical entity file), there are many ways to specify physical attributes through the ENVIRONMENT option (which is an additive attribute). The following attributes may be specified: record format (F, V, VS, VBS, U- with block size and record size), number of buffers, tape positioning information (LEAVE/REWIND), track overflow for writing on disk, and many more.

5.3.5 Opening and closing files. Before a file can be used, the statement OPEN must be used. This implies some preparatory action, like checking the availability of storage media, positioning the medium, allocating access routines and buffers. When processing is completed, the statement CLOSE is used to close the file. This involves release of the facilities that were established during opening of the file.

The syntax for OPEN is:

OPEN FILE (⟨filename⟩) ⟨option list⟩

The option list may be empty or it may specify some of the alternative or additive attributes described for the file declaration (however, ENVIRONMENT must not be used). Other options that can be used are

TITLE (⟨string⟩)

which associates the file to a data set defined by a JCL DD-card with ddname=⟨string⟩, and LINESIZE and PAGESIZE parameters.

The syntax for CLOSE is:

CLOSE FILE (⟨filename⟩)

OPEN and CLOSE are optional statements: The first data transmission statement opens the file, task termination closes it. If OPEN is omitted, the default OPEN deduces some file attributes from the I/O statement.

5.3.6 Associating file names with data sets. This is done outside the PL/1 program by DD statements in JCL (=Job Control Language):

```
// filename    DD    DSNAME = ⟨dsname⟩
or
// ddname      DD    DSNAME = ⟨dsname⟩

OPEN FILE (⟨filename⟩) TITLE ('⟨ddname⟩')
```

In each case, the data set with name ⟨dsname⟩ is associated with the file.

In each PL/1 program two standard files are defined that may be used without declaration:

```
DECLARE SYSIN FILE STREAM INPUT,
        SYSPRINT FILE STREAM OUTPUT PRINT;
```

5.3.7 I/O statements for record-oriented transmission. On input, the READ statement causes a single record to be transmitted to a program variable exactly as it is recorded in the data set; on output, the WRITE, REWRITE, or LOCATE statement causes a single record to be transmitted from a program variable exactly as it is recorded internally. All blocking/deblocking is performed automatically.

Variables involved in record-oriented transmission must be unsubscripted and cannot be parameters. The following five statements may be used (subject to restrictions of file attribute):

READ can be used with any INPUT or UPDATE file to read one record into a variable or a buffer. The forms are:

READ FILE(f) [INTO(v)] [KEY(k) | KEYTO(s)] [EVENT(e)]

READ FILE(f) SET(p) [KEY(k) | KEYTO(s)]

READ FILE(f) [IGNORE(n)]

where f is a filename, v a variable into which the record is to be read, k a string expression specifying the key of the record to be read, s a string variable into which the key will be read, e an event variable, p a pointer variable pointing to the buffer, and n an integer expression telling how many records should be skipped.

The following three statements can be used to write into a file:

WRITE FILE(f) FROM (v) [KEYFROM(k)] [EVENT(e)]

REWRITE FILE(f) FROM(v) [KEY(k)] [EVENT(e)]

LOCATE v FILE(f) [SET(p)] [KEYFROM(k)]

The last statement can be used for BUFFERED OUTPUT only. It allocates a based variable in a buffer and may also cause transmission of a previously allocated based variable.

5.3.8 I/O statements for stream-oriented transmission. The data set is treated as a continuous stream of data items in character form; within the program, record boundaries are ignored. However, a data set is considered to consist of a series of lines, and each data set created or accessed by stream-oriented transmission has a line size associated with it. In general, a line is equivalent to a record in the data set; however, the line size is not necessarily equal to the record size.

There are three modes of stream-oriented transmission:

(1) list-directed:
Transmission without specification of format. On input, the data items are character strings in the form of signed constants which will be converted to internal representation. The variables to which the values are to be assigned are specified by a data list. On output, data items are specified by expressions the values of which are converted to string representation. For PRINT files, data items are automatically aligned on present tabulator positions.

(2) data-directed:
Transmission of self-identifying data. On input, each data item is in the form of an assignment statement that specifies both the variable and the value to be assigned to it. On output, each data item is represented as an assignment statement. If a data list is omitted on output, all variables that are known within that block are transmitted. For PRINT files, the items are aligned as for list-directed output.

(3) edit-directed:
Transmission of data with format specification. On input, data are considered to be a continuous stream of characters. The variables to which values are to be assigned are defined by a data list. Format items in a format list specify the number of characters to be used for conversion of a value and details of conversion. On output the data values to be transmitted are defined by a data list. The conversion is controlled by a format list.

Data transmission statements for STREAM transmission are:

GET FILE(f) ⟨data spec⟩ [COPY] [SKIP(⟨expr⟩)]
GET STRING (⟨char-string-name⟩) ⟨data spec⟩

The COPY option effects copying of the input record to the SYSPRINT file. The SKIP option causes to start at a new line. GET STRING uses the char-string as input source. The data-spec has the following form: ⟨element⟩{,⟨element⟩}* where an element specifies a data element, an array, a structured variable, etc. The syntax is:

```
data spec  ::=  LIST (⟨data-list⟩)
             |  DATA (⟨data-list⟩)|DATA
             |  EDIT {(⟨data-list⟩)(⟨format-
                                   list⟩)}*
```

5.4 Sequential Files

In the first three subsections of this section we have looked at the ways how files are accessed in three programming languages. In this subsection we shall attempt to summarize those different approaches into one unified view of sequential files[15].

A sequential file is an ordered sequence of data elements of one or more types (in the sense of PASCAL), defined by the file declaration. The data elements are called file elements.

Since the order of the sequence is defined by the order in which the elements are created, an ordinal number is effectively assigned to each file element, starting with 1 and up to n, the total number of file elements in the file. The size of the file is left undefined and may be restricted by the environment in which the program is executed.

A file in the above sense is a logical entity with the properties described in the following sections and with a set of operators acting on the file. In general, only those properties are relevant within a program. However, the file may have some physical properties which are defined by the environment and which may provide for better processing efficiency. Specification of those features is possible in the command language environment of the information processing system on which the program is executed.

One of those environment dependent properties is the direct accessibility of file elements if the file resides on a direct-access storage device. If the implementation is done accordingly, the ordinal number may be used to access individual file elements. Therefore, in the following discussion no distinction will be made between a sequential file on a direct-access storage device and a "direct-access file". Moreover, if such a file is moved to a magnetic tape, it is only accessible as a sequential file.

5.4.1 Declaration of a sequential file. A sequential file is declared by

FILE ⟨id⟩ OF ⟨type⟩ {, ⟨type⟩}*

The declaration associates the structured type "file" with the identifier. The file elements may be of one type only or of several types, for a given file element is completely arbitrary and defined by the write-operation.

Obviously, introduction of the type

list may cause problems to the compiler when checking for type compatibility. Possibly, some checks can only be carried out at program execution time.

In the following, the file name identifier will be used in two ways:

(1) it denotes the file as an entity; operations of this type will be OPEN or CLOSE or general positioning operations; in these cases the identifier will be preceded by the keyword "FILE";

(2) the file identifier stands for a file element which is treated as a normal variable except that each occurrence may produce a new value;

The sequential file defined by the declaration above is a logical entity. A real entity - called a "data set" in some systems, or simply "file" - is associated with it. If this file only exists locally, during program execution, and if it is only accessible by the program itself, there are no problems with the program operating on that file. If, however, the file is permanent and thus sharable with other users, certain access rights of a user to a file must be taken into consideration. A temporary file, that exists during execution of a sequence of job steps, each being an independent program, takes a somewhat intermediate position: It may be desirable to restrict (possibly only for program reliability reasons) the access to the file by some of the programs.

In the following sections, a set of operations on files is defined. Not all of them are compatible with each type of access right ("read-only"-access, "write"-access, etc.).

The following processing modes for a file are distinguished:

READ file elements may only be read sequentially;

EXTEND file elements may only be appended sequentially at the end of the file; this mode is equivalent to the usual "WRITE" if the file is initially empty;

REWRITE existing file elements are discarded before any new elements are appended (i.e., after erasure of the existing information this mode is equivalent to EXTEND);

ENQUIRE file elements are accessible in random order on the base of the ordinal number defined for the element; no sequential read operations are defined (i.e., sequential read must be performed by increasing the element number by 1 each time); each file element may be read several times;

UPDATE file elements may be both read and written in any order, the rewrite being subject to the restriction that the new value must be of the same type as the existing one; in addition, file elements may be appended at the end of the file sequentially.

It seems straightforward to express access rights in terms of these processing modes. The modes READ, EXTEND and REWRITE are called sequential modes, the modes ENQUIRE and UPDATE are called direct-access modes.

It is desirable to be able to process a file in several modes within one program (provided the program has the corresponding access rights). This can be done when "opening" the (logical) file, thereby indicating the processing mode. After "closing" it, it may be reopened for another processing mode:

OPEN FILE ⟨file-id⟩ TO ⟨processing mode⟩

CLOSE FILE ⟨file-id⟩

⟨processing mode⟩ ::= READ|EXTEND|REWRITE
| ENQUIRE|UPDATE

After a processing mode has been established, the set of file operations associated with that mode may be executed. Operations not defined for a particular mode are illegal or (if consistent) are treated as "no operation". Use of the OPEN/CLOSE scheme provides a high level of security against unintentional file destruction (e.g. if a file may be both read and rewritten, the write operation is illegal if the file has been opened for READ).

The question, whether OPEN/CLOSE must be used or whether they will be performed by default by the system when the program is started (if there is only one mode defined in the command language environment from which the program was invoked), or whether CLOSE/OPEN pairs are replaced by other mode switching operations, is left to the programming language and its I/O interface routines.

The denotation for the processing modes need not be exactly as introduced above. Instead of EXTEND one could use the more familiar form WRITE (which, however, usually means "rewrite").

5.4.2 File operations in sequential modes. A file element is created by the file assignment statement

⟨file id⟩ := ⟨T-expression⟩

The statement is legal if the file processing mode is EXTEND or REWRITE at the time the statement is executed. The type of the ⟨T-expression⟩ must be assignment compatible with one of the types indicated in the file declaration. This implies that some type conversion (e.g. integer→real) may be done. If the file type is a record (in the sense of Algol W,[16] for example), the file assignment may look like

⟨file id⟩ :=⟨record class id⟩
(⟨expression list⟩)

If a file is processed in READ mode, elements may be retrieved one by one, using the file retrieval statement

⟨T-var⟩ := ⟨file id⟩

Each time a new element of the file is retrieved, and the file pointer is advanced

by one. The value of the file element must be assignment compatible with the type of ⟨T-var⟩. This implies that the program has knowledge of the type of the incoming file element. If such knowledge does not exist, the function

integer function FILE_TYPE_CASE (⟨file id⟩)

will return an integer value indicating the type of the incoming element in terms of the position of the type identifier in the type list of the file declaration. The file pointer is not changed.

The above definition of the file retrieval statement implies that a file element cannot be used in any other way. This is to some extent contradictory to the treatment of a file element as a normal variable. However, if a file element is used as any other variable - for instance (if it is not of simple type) as primary in an arithmetic expression - one gets into difficulties with arithmetic operator domains if the file has multiple types.

Since the file has no defined length (in terms of number of elements) it must be possible to anticipate the end of a file. This is possible by using the function

boolean function EOF (⟨file id⟩)

which returns the value "false" if there exists an incoming element, and "true" if there are no more elements. This function is needed only in the READ mode; in the two other sequential modes it may be considered as illegal or always return the value "true". The file pointer is not changed.

If a file has to be read repeatedly, the reset statement

RESET FILE ⟨file id⟩

resets the pointer to 1. The next retrieval statement will return the value of the first element of the file.

For all sequential modes the procedure

integer function NEXT_ORDINAL (⟨file id⟩)

is defined. It returns the ordinal number k of the incoming file element, if EOF (⟨file id⟩) = false or -n, where n is the number of elements in the file, if EOF(⟨file id⟩) = true.

It is also possible to use reference variables to point to file elements. It is assumed that a reference variable, when used with a file, is connected with only one type out of the set of types associated with a file. The reference assignment

⟨refvar⟩ := @⟨file id⟩

will set ⟨refvar⟩ to point to the next sequential file element in READ mode and to a new "empty" file element in EXTEND or REWRITE mode. (Since ⟨refvar⟩ is connected with one type only, the length for the allocation of the file element in the buffer is known).

5.4.3 File operations in direct-access modes. If a data set associated with a file resides on a direct-access storage device, it is possible to access file elements directly by specifying their ordinal number. Consequently in ENQUIRE or UPDATE mode only file assignments or file retrievals of the form

⟨file id⟩(k) := ⟨T-expression⟩
⟨T-var⟩ := ⟨file id⟩ (k)

are legal, k denoting the ordinal number of an existing element. The first statement assigns a new value to the file element with ordinal number k. This form of file assignment statement may only be used in UPDATE mode, but the file retrieval statement is legal in either of the direct-access modes.

Frequently it will be necessary to create new elements of the file. In spite of the direct-access properties the file is still organized sequentially, which implies that elements can only be added at the end (= appended). In principle, two approaches are conceivable:

(1) If in the file assignment statement the ordinal number k is equal to n+1, where n is the current file length, a new element is appended. Any other value of k which is >n is illegal.

(2) A new file element is appended by

⟨file id⟩(NEW) := ⟨T-expression⟩

where NEW is a reserved word of the language and effectively specifies k=n+1.

The two approaches are basically equivalent. (1) has the advantage over (2) that no new language keyword is required. On the other hand, (2) states more clearly the intent of the programmer to create a new file element, thus reducing the danger of accidental errors. It appears that a function to find the length of the file is needed in both cases. One could introduce a function

integer function L (FILE ⟨file id⟩)

which would return n. This function only makes sense for files residing on direct access store, where the file length can be stored in the directory of the file. For a file residing on tape, n is not known in advance.

Also when processing files in direct-access modes, reference variables can be used. According to what has been said above, the reference assignment would take the form

⟨ref var⟩ := @⟨file id⟩(k)

or, if approach (2) is selected for creating new elements,

⟨ref var⟩ := @⟨file id⟩(NEW).

It is desirable to introduce the further procedure

procedure FIND(⟨file id⟩(k))

where k denotes an existing element, or it

is k = n+1. The invocation of this procedure has no effect on the program. It only serves to provide concurrent execution of program and access to a file element.

In order to make this file system as much foolproof as possible, the file declaration (in form of a file descriptor table) should be stored away in the file directory. Each record would have stored with it a reference number to the type list, thus telling its type. In this way we would arrive at self-describing files which can be converted without difficulties from one representation (i.e. computer) to another.

5.5 Indexed Files

In this subsection we shall briefly introduce the concept of indexed files, proceeding in analogy with sequential files in direct-access mode.

An indexed file is an ordered set of elements, each element consisting of a pair of values: a key value and a data value. (An indexed file can be considered as a binary relation (key:data) or as a function defined on the key domain.) An indexed file allows accessing of file elements in both random and sequential order, by specifying (for random access) the value of the key. Ordering and accessing imply that the key type fulfills the requirements that

(1) the equality operator = is defined
(2) the relational operator < is defined

An indexed file is considered to be ordered by increasing key values. Elements may be added any time in any place, provided no element with the same key already exists. File elements may also be deleted or they may be modified in their data parts. The data part may also have the value "null" or "undefined".

5.5.1 Declaration of an indexed file.

An indexed file is declared by:

FILE ⟨id⟩ OF (⟨key type⟩:⟨data type⟩ {,⟨data type⟩}$_0^*$)

For the ⟨key type⟩ , the equality operator(s) = (and ≠) and the comparison operator must be defined. Otherwise the types are arbitrary. The same processing modes as those defined for sequential files are also defined for indexed files. Hence, an indexed file may also be processed sequentially.

5.5.2 Operations in sequential modes.

Operations in sequential modes differ from those defined for sequential files in the fact that the key must be supplied when writing and that it may have arbitrary values, except for the restriction that keys must increase in value during sequential operations.

A file element is created by the file assignment statement

⟨file id⟩ := (⟨key value⟩:⟨data value⟩)

The same restrictions as for sequential files hold, as far as processing modes and assignment compatibility for the data values are concerned.

In addition, a check is made that ⟨key value⟩ is greater than the key of the previously created file element.

Similarly, file elements are retrieved, one by one, by the file retrieval statement

(⟨key var⟩:⟨data var⟩) := ⟨file id⟩

The value of the key is placed into ⟨key var⟩, the value of the data part into ⟨data var⟩.

Analogously to sequential files the following functions are defined:

integer function FILE_TYPE_CASE(⟨file id⟩)
⟨key type⟩ function NEXT_KEY(⟨file id⟩)
boolean function EOF(⟨file id⟩)
RESET FILE ⟨file id⟩

5.5.3 Operations in direct-access modes.

In direct-access modes, file elements are accessed by their key:

⟨var⟩ := ⟨file id⟩(⟨key value⟩)

The key is used like an index. In a similar notation a file element is written by

⟨file id⟩(⟨key value⟩) :=⟨data value⟩

Sometimes it may be desirable to know whether a file element with a given key exists. The function

boolean function KEY(⟨file id⟩(⟨key value⟩))

will do the test. If a new file element has to be written and one wants to ensure that it does not yet exist, a file assignment statement of the form

⟨file id⟩(NEW⟨key value⟩) :=⟨data value⟩

can be used.

5.6 Summary

We have introduced a nomenclature for accessing files which attempts to treat files like other variables. In fact they resemble one-dimensional arrays. When comparing indexed and sequential files, we come to the conclusion that sequential files can be treated like special cases of indexed files where the key is of type integer and the keys assume all possible values between an including 1 and n, the file length. The only difference is that the key for sequential files need not be given.

Let us again stress the importance of having self-describing files - a necessary condition if files are to be portable between different computing systems.

6. Program Execution

In this section we shall look at the normal job control functions a command language must provide. It should be the aim to make the definition of jobs as simple as possible. That means that trivial jobs

should be specified with only very few lines, and that the need for additional information should increase proportional to the increased sophistication of jobs.

6.1 A simple program

Let us now consider the simplest program we can imagine: A FORTRAN program is to be compiled and executed. It will read some data and all output will go to the standard line printer. We believe the CL statements can be as simple as this:

```
EXECUTE FORTRAN PROGRAM:
   [ FORTRAN source program ]
WITH INPUT:
   [ Input data             ]
END
```

This representation implies several conventions and capabilities of the CL interpreter. They are: The interpreter is capable of detecting the line "WITH ..." as being non-FORTRAN but rather a CL statement line; the input data does not contain text and therefore the line "END ..." can be recognized as a CL statement line; there are some defaults for handling the printer output.

It may turn out that it is more convenient to have all CL statement lines start with a special symbol, like a %-sign. But in general I feel that such techniques should be avoided. The user would probably prefer to speak "English" to his computer. The above example is more or less tailored to the needs of people that use cards. It may turn out the arbitrary data must be surrounded by some delimiters (like $$$, to be specified by the user) not occurring in the data itself.

6.2 A little more complex program

Obviously many users will have their programs stored as files in the system. Even external data would not be supplied from punched cards. Instead, a user will type the data into his terminal and store it in another file. Frequently, he would also like to specify some additional parameters for the execution of his program which would not be part of the input file.

The following example is a little more sophisticated:

```
EXEC PROG analyse
WITH
     INPUT FILE data = newdata;
     OUTPUT FILE list: printer 5 (8 PAGES);
     PARAMETERS:
       title:= "experiment " || DATE,n:=3;
     FILE result = rfile(REWRITE);
     FILE tables = mastertables\mychange;
END
```

The following general rules apply: CL keywords are written in upper case, file names and other information is written in lower case. The keywords EXCUTE and PROGRAM have been shortened. The CL constructs have the following meaning: Since no programming language has been mentioned in the first line the CL interpreter will assume that "analyse" is an object program belonging to the user who supplied this CL statement. The block WITH ... END specifies the input/output parameters of the program. Input is to be taken from a file "newdata" and it will be read from within the program by using the logical file name "data". (In standard IBM/37o FORTRAN conventions it would be INPUT FILE 5 = newdata.) The output file is sent to a special printer called "printer 5" (which is remote, say) and up to 8 pages output are requested. The program permits (and requests) the specification of the additional parameters "title" and "n" . They are specified in an assignment list following the keyword PARAMETERS. The string assigned to "title" is the concatenation of the word "experiment" and the current date (DATE is CL procedure returning the current date in string form). The program will then in addition rewrite the file "rfile" and it will read data from a file "mastertables". However, the contents of "mastertables" will be updated by a file "mychange", consisting of update commands. The symbol \ is an update operator.

Being not able to present a complete CL at the present time, I have introduced loosely a way of how to describe a job to the CL environment. In the last part of this section we shall look in a little more detail at the way how files can be declared in the CL environment.

6.3 File declarations in the CL environment

In the last section we have discussed the problems of the specification and usage of files in a program. We have introduced the concept of "logical files" and we have mentioned that they will be associated with "real files" (or "data sets") during execution of the program. Establishing the correspondence between logical and real files will be part of the job description.

The set of logical file names used by a program should be considered as a set of formal parameters. Since we have to give up the concept of positional parameters in a CL, the logical file names must be known outside the program, i.e. in the CL environment.

We have a set of real files with which we shall associate the logical files. We may consider two classes of real files to exist: temporary files and permanent files.

A temporary file exists only during execution of a job and may be accessible by one or more job steps. The creation and lifetime of temporary files is controlled by the block structure of the CL program. Temporary files may be defined by:

```
DEFINE FILE ⟨filename⟩ : (⟨file attri-
                          butes⟩)
```

The filename is the name of the temporary file.

A permanent file is created when needed and deleted when not needed any longer. Permanent files exist independently of jobs. A user may create them at any time (i.e. from a terminal or by a job) using the CL statement

CREATE FILE ⟨filename⟩ : (⟨file attributes⟩)

where "filename" is the name of the permanent file when created. It may later be deleted by

DELETE FILE ⟨filename⟩

Now that we have created the (real) files we need during execution of a job, we must be able to associate real files with logical files. This may be done by a "file correspondence declaration" of one of the forms:

FILE ⟨l-filename⟩ = ⟨r-filename⟩ (⟨file attributes⟩)

FILE ⟨l-filename⟩ : (⟨file attributes⟩)

On the left-hand side we specify the logical file name, on the right-hand side in the first case the real file which is to be used. The file attributes specified for this program may be more restrictive than the attributes permitted for this user. For instance, if the attribute specified is "READ", the file may never be inadvertently overwritten. The second form is an association between a logical file and a special form of temporary file which we will call "local file". This is a file without a name (or, without a name known to the user) and specified only for this program. We will also call it a "local file". By default it can be written and read and the main attribute to be specified is its length. For a FORTRAN program, where logical files are specified by their "file reference number", the correspondence declaration could look like:

FILE 2 = myfile (READ)

FILE 3 : (500 K)

It may happen that a programmer wants to concatenate the print output from several programs (job steps) into one print file. He would then define a temporary file, "printfile", say, and then use the first of the above forms:

FILE 6 = printfile (EXTEND,500 LINES)

at the same time specifying an upper line limit for the extension.

In the job description of the previous subsection we have used special forms of file definitions: input files and output files. One could use the following syntax

INPUT FILE ⟨filename⟩: Ⓜ ... Ⓜ

INPUT FILE ⟨filename⟩: ⟨terminal id⟩

OUTPUT FILE ⟨filename⟩: ⟨printer id⟩ (⟨options⟩)

to specify details about the input and further processing of the output. Ⓜ are user defined delimiters to isolate arbitrary input data from the CL statements.

7. Embedding Programs into the CL Environment

In this section we will discuss the embedding of user programs into the CL environment.[17] It will be shown that many of the necessary mechanisms may be developed by starting from some well-known techniques used for procedure implementation in some high-level programming languages.

Before we do this, however, we will establish our terminology by explaining the meaning we want to attach to the words "algorithm", "program", "computation", and "process". This is necessary because those concepts are frequently used with at least partially overlapped meaning and somewhat fuzzy definitions.

7.1 Algorithms, programs, and computations

By algorithm we understand an abstract formulation of a solution to a computational problem which can be formulated and executed in a finite number of steps. In its simplest form an algorithm maps an input domain of data into an output range of data. We will assume that an algorithm behaves decently, i.e. it terminates after a finite number of steps and does not "blow up", like producing an array overflow or a division through zero, etc.

In its simplest form such an algorithm may be represented by an Algol procedure, taking some inputs and producing some outputs as defined by its formal parameter list. The procedure is described by a body of Algol statements and may or may not require some additional workspace to work with. Such a procedure cannot "run by itself" but has to be embedded into a program which provides the environment for its execution.

The procedure mechanism has been described in such detail because it provides an analogy to the execution of programs. We shall call program the description of a given solution to a data processing task in terms of a sequence of "instructions" or "statements", specified in a suitable programming language (like machine code for a "real" computer or Algol for an abstract "Algol machine").

The definition of program is very similar to the definition of algorithm. However, there is one important difference: in a realistic CL environment we must be able to treat all kinds of programs including those where the implementation of the intended algorithm failed, thereby leading to incorrect programs that get into a loop or violate storage bounds or "blow up" in any other way. The CL environment has to provide a safe way of executing such erroneous programs and preventing the "spreading" of such fatal errors.

A computation is the actual execution

of a program on a computer. What we have said above about possibly erroneous programs comes into effect when those programs are executed on a computer. While such a program is executed we consider the computation embedded into the CL environment. The effectuation of the program execution is considered to be a CL procedure invocation. The static embedding of the program to be executed is analogous to a procedure embedded into an ordinary PL program. By the term "CL environment" we denote the actual, dynamic embedment of the program execution into the operating environment (the operating system). In short: the CL provides the definition and description of the interface between the user's program and the operating system.

7.2 PL/CL interface, resources and the operating system

We may describe the relation of a computation to the CL environment on three levels: the specification of the interface in terms of CL/PL constructs, the utilization of resources, and the operating system.

(1) CL/PL interface:

- the program describes the computation, i.e. it defines the course of actions to be taken by the computing system during the computation;
- there are some input variables passed on by the CL environment to the computation just before it is started (including some input files);
- there are some output variables returned by the computation to the CL environment immediately after it has terminated (including some output files);
- there is a (possibly empty) set of shared variables which are shared with other computations or the CL environment; these variables may be accessed during the computation;
- there is a (possibly empty) set of files the computation has access to; they may be temporary or permanent.

(2) Resources:

- there is a processor (usually called a "CPU") which is able to carry out the computation ("execute the program");
- there is a work store (memory) into which the program is loaded and from which space is allocated for intermediate results;
- there may or may not be some other resources the computation has access to;

(3) The operating system:

- there is an operating system which controls the CL environment and the initialization, execution and termination of the computation (including the assignment of resources like work store space, processor time, and possibly other resources);
- more specifically, there is a process control block which establishes the embedding of the computation into the operating environment; this PCB in general is not accessible by the computation.

The user of the computing system is concerned with those three groups of entities to a different degree. The items of group (1) have to be described in much detail. By the normal user, resource requirements need to be specified in general terms only (like core space or CPU time); any more detailed requirements are necessary only for the "computer specialist" who is concerned with the operating system itself. Finally, regarding the operating system, there are only very few parameters that may be specified. Typically, one might think of the accounting information which is associated with the PCB and where some options may be specified by the user.

We conclude that a CL must provide the possibility of defining procedures in any kind of language (including CL itself, of course), since programs may be written in many different languages (e.g. assembly language, FORTRAN, Algol, PL/1, etc.). This is in contrast to procedures in normal programming languages where only in exceptional cases a procedure may be written in a language other than that of the invoking environment.

7.3 Access modes to CL variables

One can distinguish between several modes of transmission of information between CL environment and computation:

(1) initialization parameter to program ("value" parameter) transmitted when computation is started;

(2) termination parameter from program ("Result" parameter), being transmitted to CL environment upon termination of the computation;

(3) initialization/termination parameter ("value/result" parameter), a combination of (1) and (2);

(4) shared variables and synchronization variables which may be used throughout the computation. There may be several kinds to be described later.

The similarity of these parameter transmission modes with constructs in programming languages is evident. The best example is ALGOL W with VALUE, RESULT, VALUE RESULT and name parameters. It should be noted that most existent job control languages only have parameters with modes (1) or (2).

7.4 Shared variables as a means for synchronization

Shared variables and synchronization variables can be of various types. Because they are required to be accessed any time,

some elaborate access mechanisms must be set up to ensure their integrity and to maintain the protection formulated by the access rights. We shall distinguish the following types of shared variables:

(1) dynamic input parameter which can be modified in the CL environment and can only be read by the programm; in order to avoid inconsistency the parameter must be totally transmitted in one indivisible operation (for simple variables a normal read or move will do); there may be at most one process that sets the parameter, but any number of processes may read it;

(2) dynamic output parameter which is used to transmit information from the computation to the CL environment; the computation is the only process that can set the parameter, though many others may read it.

(3) shared variables can be of any kind (single variables, arrays, records, files, etc.) and are accessed with the help of the synchronizing primitives "mutex statement" and "conditional mutex statement" (mutex = mutually exclusive)[18]. Any number of processes may share such a variable (file sharing may be accomplished through a set of suitable file processing modes available to the user);

(4) semaphores are accessed through the synchronizing primitives "wait" and "signal".

(5) event variables exist in various operating systems and are accessed through the synchronizing primitives "await" and "post". Any number of processes may be made to wait for the event; a "post" issued by one process releases all of them.

(6) message queues have been used in the RC4000 operating system but also in some time-sharing systems.[18] The synchronizing primitives are "send message", "receive message"; it probably would be appropriate to consider message queues as special cases of shared files;

We have listed a set of different shared variables or synchronization variables, like some of these appear in the literature. Each one will in fact be represented as a highly complicated data structure, together with a set of operations that may use it.

In general, a computation has no access to the CL environment. The only exception are files and the parameters we are discussing here. For the reason of access restriction and protection, shared variables must be kept outside the realm of computation. Controlled access is then achieved through a combination of

(1) expressing access requests to shared variables through a call to the supervisor of the operating system; the supervisor will carefully check the access authorization of the computation;

(2) copying the shared variable into the work area of the computation, where the computation has free access to it (in such a case a lock will prevent any more copies from being made); when the shared variable is no longer needed, it will be copied back and unlocked.

This mechanism is not only a safeguard against illegal use of information, but it also yields another advantage: the CL environment and the computation (or generally speaking, a process) need not be on the same computer. Access requests and the copy of a variable can be performed between two connected CPUs or they can be sent over telecommunication lines.

7.5 Program variable/CL variable correspondence

Let us assume we have a program written in a programming language that permits the specification of shared variables. It is convenient to state within the program the access modes under which the variable is going to be used.

When a program is envoked from the CL environment, the program variable names are treated similar to formal parameters of a procedure call. CL variables, which will be the actual shared variables, must be named in a parameter correspondence declaration. This declaration may take the following form:

PARAMETERS: ⟨parm corresp⟩{,⟨parm corresp⟩}$_0^*$

⟨parm corresp⟩ ::= ⟨id⟩ :=⟨expr⟩
| ⟨access mode⟩ ⟨id⟩ =⟨var⟩

⟨access mode⟩ ::= RESULT | VALUE RESULT
| SHARED | INPUT | OUTPUT
| SEMAPHORE | EVENT
| MESSAGE

⟨id⟩ is the name of the parameter in the program, ⟨var⟩ is the name of a variable in the CL environment. Explicit access mode indication is omitted in the case of a value parameter. Instead, an assignment form of correspondence is used.

The access restriction requirement demands that the access mode given in the CL environment in conjunction with program invocation supercedes the access mode specified in the program. Specifically, if no access mode is specified in the program (maybe because the PL does not permit it), the CL declaration takes full responsibility.

The above syntax is equivalent to using "key-word" parameters, in the sense that the order in which the equivalences are indicated is immaterial. A parameter not indicated will be assumed non-existent. (If it is accessed in the program, it may be treated as an error or the program may just use the internal copy; the latter way of doing it may be useful for test purposes).

If a programming language does not provide declarations for external variables,

one might like to resort to positional parameters. The syntax could be:

⟨parm decl⟩ ::= PARAMETERS: ⟨parm⟩ $\{ , \langle parm \rangle \}_0^*$

⟨parm⟩ ::= ⟨expr⟩
| ⟨access mode⟩ ⟨var⟩

The computation, when started, will have to use a special procedure call to establish the correspondence between the CL variables and the internal variable.

7.6 Establishing access to CL shared variables

We have introduced four kinds of access modes to general shared variables. Semaphores, events, and message queues are treated directly by their synchronizing operations. Depending on the mode, the system has to perform certain actions when the computation is initiated, while it executes and when it is terminated. This can be seen from the following table ("V" is the variable in the CL environment, "V" a variable in the computation environment):

access mode	init	exec	term
VALUE	v := V	access to v	-
RESULT	-	access to v	V := v
VALUE/ RESULT	v := V	access to v	V := v
SHARED	establ access path	access function	release V if locked

The following table lists the shared variables and the corresponding access or synchronizing operations:

shared variable	access functions
INPUT	GETPARM
OUTPUT	PUTPARM
SHARED	RESERVE/RELEASE or MUTEX block
SEMAPHORE	WAIT/SIGNAL
EVENT	AWAIT/EVENT
MESSAGE QUEUE	RECEIVE/SEND

The discussion in this chapter was to show what direction further development of Command Languages could possibly take in order to solve the problems of communicating processes. These problems have to be solved if we are going to construct more sophisticated computing systems in the future.

References

(1) P.H. Enslow
Operating System Command Language: A Brief History ot their Study Command Languages, pp. 5-24, North Holland Amsterdam (1975)

(2) D.W. Barron, I.R. Jackson
The Evolution of Job Control Languages, Software 2, pp. 143-164 (1972)

(3) T.A. Dolotta, C.A. Irvine
Proposal for a Time-Sharing Command Structure, IFIP 1968, pp. 493-498, North Holland (1968)

(4) C.I. Stephenson
On the Structure and Control of Commands ACM SIGOPS 7.4, pp. 22-26 & 127-136(1973)

(5) Project MAC
The MULTICS Programmer's Manual, MIT Report, Nov. 197o

(6) S. Lauesen
Program Control of Operating Systems, BIT 13, pp. 323-337 (1973)

(7) I. Jensen, S. Lauesen
Programming Language Extensions which Render Job Control Languages Superfluous, Command Languages, pp. 137-152, North Holland, Amsterdam (1975)

(8) I.T. Parsons
A High-Level Job Control Language, Software 5, 69-82 (1975)

(9) R.F. Brunt, D.E. Tuffs
A User-Oriented Approach to Control Languages, Software 6 (1976)

(1o) G.D. Brown
System/36o Job Control Language
J. Wiley, New York (197o)

(11) C. Gram, F.R. Hertweck
Command Languages: Design Considerations and Basic Concepts Command Languages, pp. 43-69, North Holland,Amsterdam(1975)

(12) N. Wirth
The Programming Language PASCAL, Acta Informatica 1, pp. 35-63 (1971)

(13) The FORTRAN IV Language,
IBM Manual GC28-6515-1o(1974)

(14) The PL/1 Language,
IBM Manual GC28-82o1-4 (1972)

(15) F.R. Hertweck
An Approach to a Unified View of File Handling, IPP Report R/8

(16) N. Wirth, C.A.R. Hoare
A Contribution to the Development of Algol, CACM 9, 413-431 (1966)

(17) F.R. Hertweck
Computations and the CL Environment
Report IPP R/9 (1975)

(18) P.Brinch Hansen
Operating Systems Principles
Prentice-Hall, Englewood Cliffs (1973)