operating system itself for file access. Finally, the user must have available the facilities that the language processors use in accessing files.

5. Conclusions

I have tried to outline a few of the problem areas of operating system interaction with statistical computing. However, many problems went unmentioned and many pitfalls exist for which I have not given adequate warnings. There is currently no substitute for practical tests. However, if the operating system operates efficiently using only those facilities which it provides to the user then there is some chance that user programs may be able to operate effectively. If the system is inefficient or if it has to rely on special facilities which are not available to the user, then there must be considerable doubt about its usefulness.

The simplest question one can ask to establish the possible usefulness of an operating system is: Would you be comfortable about the prospect of writing a powerful and flexible operating system using the operating system? If the answer to this question is "no" then the chances are good that attempts to use the system for serious statistical computing tasks will encounter repeated frustrations and restrictions.

Reference

Nelder, J. A. and Cooper, B. E. (1971). Input/output in statistical programming. *Appl. Statist.*, **20**, 56–73.

Input/output in Statistical Programming

By J. A. Nelder and B. E. Cooper†

Rothamsted Experimental Station *Atlas Computer Laboratory*

Summary

Input/output (I/O) is analysed in terms of the transfer of items of various types having various internal and external representations between various internal and external positions. Ways of describing representation and position are discussed, for both character and binary data. Three basic properties of good I/O facilities for statistical computing are stated, and the facilities offered by four existing programming languages are matched against them. Finally, extensions to Fortran are proposed that would provide the properties sought.

Introduction

We suspect that input/output (I/O) operations cause more trouble than any others to computer users. Making the internal algorithms (not concerned with I/O operations) work is often much easier than reading in the data they require, or printing the results. No doubt the I/O operations are inherently messier than the internal operations, but much of the difficulty arises from deficiencies in existing high-level programming languages. I/O operations may be considered as part of a wider class of operations (called "transput" in the Algol 68 Report) concerned with the transfer of information between parts of a computer and its peripherals. We shall be concerned with the pure I/O operations of reading information from an input medium into core and of outputting information to an output peripheral such as a line printer or card punch. Much

† Now at ICL Dataskil Ltd, Reading.

will be relevant to the more general class of transput operations involving transfer to and from backing store. Having described the facilities we would like, we shall discuss how far these are provided in Fortran, Algol 60, PL/1 and Cobol, and indicate what extensions are needed to the commonest of these, Fortran, to give the I/O flexibility required. Algol 60 in its pure form does not recognize external data at all; this has led to an unfortunate proliferation of I/O conventions in different implementations. We restrict our discussions to the IFIP proposals (1964), and to the Knuth report (Knuth *et al.* 1964).

1. The Basic Mapping

Input and output is concerned with the conversions between the *external* representations of data items on I/O media and *internal* representations within the computer. In external representations, as on a punch card, a data item is usually an ordered sequence of characters of various types such as digit, letter, bracket, punctuation mark and so on. Thus externally an integer is an item consisting of an optional + or − followed by a sequence of digits. A user will think of characters in their printed form but to the computer a character will be represented as a binary number transmitted by an input device or to an output device. A second possible external representation is a binary form with a single bit represented as the presence or absence of a hole on paper tape or cards, or their equivalent on magnetic tape or disc. Thus a card becomes an array of bits with dimensions 80×12. Binary data can be recorded directly, for example using Port-a-punch or mark-sensed cards, and data-logging devices often use paper tape to represent binary data.

Internal representations of data are, of course, basically binary, but can be patterned in various ways. The basic element is a *word* consisting of an ordered set of bits, varying in number in different machines from 8 to 64. Fractions and multiples of words may be directly addressable by hardware, giving the user a choice of length of storage unit. (Example: Machines of the IBM 360 and ICL System 4 ranges have addressable storage units of length 8, 16, 32 and 64 bits.) Internal representations include binary integer, floating-point number, character string, bit string and a binary-coded decimal number. Internal representations may need more than one word, for instance a double-length floating-point number or a character string too long to fit into one word.

Input procedures take a data item in an external representation and convert it to an item in an internal representation. Output procedures reverse this. The four basic parameters of either procedure are classified by a 2×2 table which is illustrated below for J in the Fortran statements for the input of three integers

```
    READ (6, 1) I, J, K
1   FORMAT (3, I, 4)
```

	representation	position
external	integer (character)	columns 5–8 of the current input record of stream 6
internal	integer (binary)	address defined by identifier J

2. The Specification of the Mapping Parameters

We now consider the specification of the mapping process, dealing with various items and their representation and position both externally on an I/O medium and internally in computer store.

2.1. *External Type and Representation*

As indicated above both character and binary representation can occur externally and we consider first the way characters are built up in sequence to form items and later how their types and representations may be expressed.

2.1.1. *Character representation*

An item is built up from a sequence of characters following assembly rules defining the context in which each character is valid. Thus a decimal point is valid in a sequence of characters expressing a number unless one has already appeared. Usually these rules are buried deep in the input routines of the language and the user is unable to define them for himself. Cooper (1968) describes a basic Fortran subroutine for the input of words, numbers and special characters punched in free-field form. A card is read as a sequence of characters and items are then built up from the sequence using an array of integers that classify the characters into groups and another array of decision integers to define the rules. The user may not only define his own assembly rules but change them during execution, and so retains full control over them.

In addition to being able to define item-assembly rules the user must be able to define new types or new representations of existing types. Standard types such as integers may be defined once and for all, but other less common types are sometimes needed and should be definable. To give the flexibility required for all but the simplest kinds of programming, a language must allow the manipulation of characters as a basic type. It is then straightforward to build up new item types or representations within the language itself and to arrange for their input and output. The simple solution is to allow type *character* in the language; this has been done in several dialects of Fortran (for example, CDC 3000 Series Fortran), in CPL and Algol 68. Unfortunately, this simple solution has been confused by the definition of strings in Algol and PL/1, a compound concept which in our view would be better expressed as an array of type character. Where type character is lacking, an alternative is to provide packing and unpacking functions that allow characters stored in words to be accessed singly. ASA standard Fortran allows (using A format) the packing and unpacking during input and output but the same facility is not available for accessing single characters internally. PL/1 does allow such access because the I/O facilities can be applied to arrays of characters stored internally.

2.1.2. *Layout characters*

In addition to assembly rules special rules are required for the layout characters such as *space*, *newline* and *newpage*. For example, the rule might be that spaces are always legitimate in items whose internal representation is a character string, but otherwise they are invalid after an item has begun, and symbols for *newline* and *newpage* are always invalid. (Note that this is not the rule for Algol programs, where, in the reference language and most implementations, layout characters are ignored.)

2.1.3. *Zero and sign suppression*

Some languages (for example Cobol and the Knuth I/O for Algol) provide variants for numbers giving options for zero and sign suppression on output and decimal suppression on input. Suppression of leading zeros is normal in scientific work, but may be undesirable in some commercial work, for example in figures on a cheque, if it leads to blanks between the £ sign and the first significant digit. The + sign in a positive number may be allowed or suppressed (the usual option). On input decimal points and exponents characters may be implied and their position indicated in the format for the item.

2.1.4. *Description of external type*

Having described kinds of external types, we now turn to ways of specifying them. External type is specified in two basic ways, (a) by type name and (b) by "picture". In the first the item is described as "integer", "string", etc. (using some convenient abbreviation), and in the second each character is categorized. For example

$$-ZZZ999$$

represents a six-digit signed integer of which the first three digits may be suppressed zeros. In both methods the description of type tends to be fused in practice with the description of position, e.g. in Fortran

I4

means an integer occupying the next four positions. The picture mode of description is theoretically the more flexible, but it can be tediously verbose in practice. Where possible item types are few and well defined the first method has considerable advantages.

A kind of input we have found useful does not depend on the type of the item being known at compile-time. Thus rather than say "read an integer in the input stream", we may wish to say "get the next item from the input stream", information on its type being returned after discovery. Interrogatory input of this type is commonplace to compiler writers analysing text, but has not been provided by any of the general-purpose languages.

2.2. *External Position*

The second parameter of the I/O process is the position of the external item. Considering again character rather than bit strings, there are two basic ways of describing position, by *sequential* or *direct* access (often called *random*). In sequential access we work through the string systematically, leaving, after each item has been assembled, a pointer to the next character in the string. The next instruction to read an item continues assembly, specifying either the field width (fixed-field mode) or relying on the first alien character to stop the process (free-field mode). In direct access the starting point of an item must be specified directly, for example by setting a tab (the *T* format of some Fortran dialects). In practice input/output strings are grouped into *records*, either explicitly or implicitly, for ease of handling. Full direct access then requires the ability to skip forwards or backwards between records. With "one-way" peripherals like card-readers and punches (record = card) or line printers (record = line) this is impossible. With magnetic tape or disc it is possible but may be inefficient.

Note that context-dependent modes of defining external position (e.g. "the integer after the next $ sign") can be reduced to one of the two basic modes by a suitable definition of items to be read (e.g. "read sequentially next character until $ found, then read integer").

To specify external position the programmer must have access to the pointers to characters in the current record and to records in the current file. This is further discussed in Section 5.3.5. below.

2.3. *Internal Representations*

Input/output operations are simplified by direct correspondences between the internal and external representations. If to every external item there corresponded an internal item *definable in the language*, the specification of internal type would be trivial. It would only be necessary to declare a suitable item and give its identifier as the parameter of an I/O instruction. Such correspondences exist in Fortran for integer, real, double-length real, complex, and a character string packable into an integer or real word. Problems arise where the correspondence fails. As mentioned in Section 2.1.1 much can be done by having type character in the language and by using arrays of characters to match external character strings. Binary-coded decimal numbers represent an internal representation available in Cobol but not in Fortran. Where an item is compound, i.e. needs several internal stores, then the definition of internal type becomes part of the problem of structure definition (see Gower and Hill, 1971).

2.4. *Internal Position*

Internal position is specified by an identifier in the broad sense, e.g. it may be a scalar (X), or an element of an array ($X(3, 4)$), or an element of a Cobol or PL/1 structure (EMPLOYEE.NET_PAY). The facilities available are defined by the general means of referencing allowed in the language and so fall outside our particular area.

2.5. *Binary Data*

In practice hardware facilities for the I/O of binary data are usually much more restricted than those for character data. Often peripherals are found where binary input or output is impossible because of parity conventions. We find this situation strange, because binary external representation is closest to the internal form of storage, and it might be thought, therefore, that the transfer would be at its simplest for binary data. Assuming for the present that there is hardware for binary input/output we consider first cards and paper-tape. The MVC system originated by A. Colin and described by Singh (1968) allows the external representation of binary variables at individual hole sites on a punched card.

2.5.1. *Binary data—external representation*

Binary data on cards or paper tape form physically a *two-dimensional array*, as distinct from the one-dimensional string of character data. Therefore an implied ordering of these dimensions must be established for the mapping process. It is probably best to take the analogous character representation with "between-characters" as the slow-moving index and "within-characters" as the fast. Thus a card becomes an 80×12 array with the second index moving the faster.

2.5.2. *Binary data—Internal representation*

Just as the manipulation of character data is helped by having type character in the language, so binary data are easily dealt with by having type *bit*. Type bit implies the storage of external bits (e.g. individual holes on a card) as internal bits and therefore differs from type logical (Boolean) which is usually implemented by filling a whole word with zeros or ones; arrays of type bit should be stored continuously in consecutive words and appropriate accessing functions should be available.

Given type bit, the I/O of binary data is simple, as is its internal manipulation. It would be interesting to know how many times type bit has been laboriously implemented "by hand" by Fortran programmers; its implementation at the compiler level is strictly elementary and it is surprising that no version of Fortran known to us contains it.

3. Practical Requirements for Statistical Work

Sections 1 and 2 describe the characteristics of the mapping underlying the I/O process. We now turn to the practical requirements for statistical work and discuss three basic properties of good I/O.

3.1. *Three Basic Properties of Good I/O Facilities*

We propose the following:

1. The basic operations deal with the transmission of single elements in a peripheral-independent way.
2. The getting from and putting to the peripheral of character strings is dissociated from their encoding/decoding.
3. The programmer retains full control when the matching of the intended with the actual item breaks down.

Property 1 implies that facilities exist for transmitting individual items of varying type, for example, single character, integer, real F format, real E format, identifier or text, with flexible format description, and that all complex I/O instructions are built up from these, *using the ordinary facilities of the language*. This requirement conflicts with, for example, Fortran, where complex I/O instructions are built up using I/O lists, implied DO loops and FORMAT statements, all aspects of the language *not occurring anywhere else*. Thus the user is forced to learn what is virtually a new set of conventions for outputting an array as distinct from calculating it. By contrast the ingenious proposals of Knuth *et al.* (1964) for Algol 60 are standard Algol but they have their own disadvantages. To mimic the I/O lists of Fortran, clumsy-looking list procedures are introduced. Compound formats produce difficulties associated with passing variable numbers of parameters that could be avoided were the single item taken as the basic element to be transmitted. We think the acceptance of the single element as the unit of transmission would simplify compilers, and give the programmer greater flexibility with less to learn, without reducing the readability of programs.

The compiled form of the compound I/O statement in languages such as Fortran and PL/1 includes calls of various input and output subroutines. These deal with the encoding or decoding of a single item and the transmission between buffer stores and I/O peripherals. To give the programmer access to these is but a small step. They are available to the user of Fortran on Atlas, but this seems exceptional. As much peripheral independence as possible is a most important requirement of the I/O routines. Thus users should have the choice of output medium at run time, and the range should include line prints, magnetic tape, disc area, cards and paper tape, and plotters and displays as available. The configurations of some devices cause difficulties that require solution, for example a line printer has a fixed number of characters per line and a new-page request is meaningful. New-page requests could be stored in an agreed form and obeyed only with a suitable output device. Information output on magnetic tape, for example, might be ultimately destined for printing, when new-page requests would become relevant.

Property 2 is widely recognized to be desirable and is implemented in many dialects of Fortran. The *READ* operation is split into *BUFFER IN* (which gets the next "recordful" of characters and stores it in an intermediate form) and *DECODE* (which assembles the required items into binary). The *WRITE* operation is similarly split into *ENCODE* and *BUFFER OUT*. This splitting has many advantages: for example it allows the remaining format of a record on input to be determined by the value of an item already encountered in that record; it allows a line of output to be built up step by step and then transmitted when complete; it allows re-scanning of a record as required. The programmer can build up procedures with whatever conventions he likes. For example, he could arrange for buffering-out to be automatically triggered by the encoding of a new-line symbol, or the attempt to encode an item into an overfull buffer.

The user is poorly served in all the common languages in respect of property 3. Originally Fortran, for example, caused a job to be abandoned when a single invalid character was met, thus allowing only one error to be found on each run. Later extensions (e.g. 360 Fortran) allow a label to be set specifying the next instruction to be obeyed when an error is found. This generalizes the label 0 facility of EMA and is an improvement, but it is still not enough. It is essential that when an error is encountered the programmer has access to what has been found in the buffer when the item was being decoded. This means knowing the current position of the pointer in the buffer, the total number of characters scanned, the total number of non-blank characters found, and the current value of the item when decoding stopped. Only with this information can the programmer have full flexibility in writing procedures for recovery from error. It is almost equally important that this information be available *when no error occurs*. For example, suppose that all columns on a data card are supposed to be punched then an error should be recognized if any columns are blank. This cannot be done unless access to the number of non-blank characters is available. Fortran, for example, often makes no distinction between a blank field and a field punched with one or more zeros. In Section 5 we consider in more detail how these desirable properties might be provided by extensions to the Fortran language.

4. I/O Facilities in Four Existing Languages

We now consider what I/O facilities are available to the programmer in four commonly used high-level programming languages, Fortran, Algol, Cobol and PL/1; and also to what extent they meet the requirements discussed above.

4.1. *Fortran*

Fortran is well established as a programming language for scientific work. Many dialects exist, most of which are standard Fortran plus additional features. The internal types of standard Fortran are integer, real, double precision (real), logical and complex. That is, types character, word, or bit are not included although a 'word', consisting of a machine-dependent number of characters, may be stored in any other type (except perhaps logical) and used in certain instructions. Type character may be simulated in store, somewhat wastefully, by storing one character per word, but packing and unpacking operations for words are not available.

The I/O operations of Fortran are concerned with a record such as a card or a line of printed output rather than with a single item. An input operation comprises the two stages of reading the next record into a buffer and decoding it to form the input items. An output operation consists of the corresponding two stages of encoding items into the buffer store followed by transmission of the buffer store to the output medium. It is not possible, within the Fortran language, to separate these two stages for either input or output.

The external types of Fortran are basically symmetric for input and output but there are some minor differences. Input and output types are:

Characters	*A*	and	*H*
Fixed point	*F*	and	*G*
Floating point	*E*	and	*G*
Double precision	*D*		
Integer	*I*		
Logical	*L*		

The type of an item to be input or output must be known at compile time. On input the types and external representations of items are specified in a format statement and faulty external representations produce errors. On output the types and external representation are similarly specified, but it is assumed by the output routines that the internal types are consistent with these.

External positions are specified in terms of field widths starting at the first column of the next record of the I/O stream. It is possible to skip columns in a forward direction but not backwards, although several dialects allow the latter, using format fragment $-nX$. The field widths specified in a format statement may not be parametric, though again some dialects allow this.

The Namelist facility, in which items are labelled on the external medium (input or output) by their internal program names, is available in several dialects and provides a restricted form of free-field representation. Otherwise free-field representation is not part of the language. For example, it is not possible to specify the input of the next integer in the current or next record without specifying exactly where it is to be found. Still less is it possible to request the input of the next item whether it be integer, real number, word or whatever.

The slight asymmetry between input and output noted above comes from two sources. The first is the special treatment of column one on output. This is used as a printer control if the output stream is directed to a line printer. Thus the format used to input items may not be used for output if column 1 of the original input record was non-blank. The second source is that *E*, *F* and *D* output forms are more rigid on output than input. For example, a decimal point is always included on output but not necessarily on input and the exponent in *E* and *D* forms uses a fixed number of columns on output. However, it is always possible to output a list of items and re-input them later using the same format statement.

Error monitoring is not specified as part of the language and an error usually terminates the run. This does, however, differ from one implementation to another and some have acknowledged that the programmer should be able to define what action be taken. We strongly endorse this view.

4.2. *Algol*

Some basic proposals for I/O in Algol 60 were laid down by Working Group 2.1 of IFIP (1964). In the procedure *insymbol* and *outsymbol* the authors proposed a way of associating the integers 1, 2, 3, . . . with a declared string of characters, and hence of reading the next character from an input stream and converting it to an integer in a declared location, or conversely for output. The procedure *length* allowed the length of a declared string to be accessed. After these very basic procedures the group went on to propose the more restricted procedures *inreal, outreal, inarray* and *outarray*. It is not clear why they proposed special procedures for one Algol type, namely *real*, while not doing so for the others, *integer, string* and *Boolean*. However, their examples show how this might be done using *insymbol, outsymbol* and *length*. The report has nothing to say on the handling or description of characters for format control like "newline" and neither does it define any standards for the external representation of items.

The Knuth report represents a greatly extended effort on Algol I/O, and was produced by a subcommittee of the ACM Programming Languages Committee. They stress that their I/O proposals "do not involve extensions or changes to the Algol 60 language". It is a matter of opinion to what extent their determination to stay within the original Algol 60 specification has hampered them in formulating their I/O proposals. In their format syntax, for which they use Algol strings, they have been heavily influenced by the picture formats of Cobol. The problem of variable format has been overcome by marking the variable replications with the character X and setting the necessary values as further parameters after the format.

I/O lists, such as occur in Fortran, are provided by means of list procedures, a device in Algol that has the effect of allowing the user to give a name to a list of identifiers (it has, of course, many other uses). The result is the occurrence of procedures such as *outlist* and *inlist* whose parameters are a channel number, a layout procedure name, and a list procedure name.

If implemented in full the Knuth proposals would give very powerful facilities to the user, yet at various critical points they fall short of the ideal.

(1) A major lack is any discussion of data structures required by a modern operating system. The concept of a record as an ordered set of items is not introduced, and so its relation to the physical structure of, for example, a line on a line-printer is not discussed. The Committee proposes that an item shall not be split over two or more lines, with which we agree, and the reasoning behind this is probably connected with an implicit recognition of the record structure of files to be input and output. By using list procedures they come close to defining a record format, such as is done explicitly in Cobol and PL/1, but do not consider how data are to be transmitted in the system. In taking this attitude they are being faithful to the spirit of the originators of Algol; however, it is arguable that in discussing I/O such an attitude has its limitations.

(2) The matching of input strings with the format is defective. First, the action to be taken when an item (say a number) does not match is undefined; this is quite unacceptable for practical use. Secondly, the occurrence of a character string in the format implies the skipping of its length in the input string, i.e. no matching at all is done. This prevents the use of character strings as switches in controlling input, a most important facility in any sophisticated programming.

(3) Binary I/O (called "non-format") is mentioned only very briefly, and the specification lacks detail for a practical implementation. The absence of type bit from the language is a difficulty here.

(4) Operations called *put* and *get* are mentioned for temporary storage of data on backing store, but again so briefly as to beg many practical difficulties in their implementation. It is not even clear if character or binary transmission is implied, but presumably the latter is intended. If so the usage is opposite to that of PL/1, in which put/get is used for characters and read/write for binary transmission.

(5) The handling of characters by packing into words (their *A* format) is more machine-dependent then it would be if type character were implemented in the language.

In spite of the undoubted power of the facilities proposed we remain convinced that a much simpler set of procedures concerned with the transmission of individual items would give the programmer all he needs, but that if general list facilities are to be provided, then explicit recognition of file structure is essential for effective implementation.

4.3. *Cobol*

Cobol was developed for commercial data-processing, and the source code is intended to resemble written English. The scientific programmer is likely to find it exceedingly verbose. The internal types (*USAGE*) of Cobol include character string (external decimal in Cobol jargon), packed decimal, binary integer, real and double-real. Packed decimal representation is almost unknown to scientific programmers. It requires special hardware for rapid internal operations and its only advantages seem to be (1) the possibility of rapid conversion (encoding) for output and (2) that it allows the exact storage of decimal numbers without rounding error. There is no type character or type bit, though editing operations are available as *EXAMINE* and *TRANSFORM*.

The external types (*PICTURE*) in Cobol are strongly asymmetrical relative to input/output. Input types are few, output types complex. Fixed fields are used throughout, and field widths may not be parametrized. Input types may consist of

(1) Alphanumeric string (*X* format),
(2) Alphabetic string (with space) (*A* format),
(3) Numeric string (9 format).

The numeric string format can be enlarged by using in the picture description *S* (for sign), *V* (for an implied decimal point) and *E* (for an exponent in a floating point number).

Output types are considerably extended, principally by inclusion of facilities for printing money values in a variety of ways. Thus the picture description can allow suppression of leading zeros, or replacement by asterisks for cheque protection, insertion of commas, decimal points, or spaces, insertion of the currency sign either in fixed position or immediately before the first significant digit, options for signs ($\pm$) and for replacing zero by blanks. The asymmetry of I/O will not allow items output by most of these special formats to be read in again.

The external position of items is governed by the original definition of the record containing them. As mentioned above, formats are restricted to fixed fields with lengths fixed at compile time. There is no provision for multiple scanning of an input record or for variable format depending on, say, the first item read.

Encoding and decoding do not enter explicitly into the language description, though the operations may take place. They appear asymetrically. Decoding occurs automatically under *READ*, which, as in Fortran, fuses the getting of the next record, and its decoding according to previously specified formats. Note that the *USAGE* option *DISPLAY* results in internal storage by characters. The *MOVE* statement (assignment) allows explicit decoding of an item stored in *DISPLAY* mode to one in binary mode. Similarly the converse allows explicit encoding. However, an asymmetry arises because files for printing must have all items in *DISPLAY* mode, i.e. ready encoded. Thus the Fortran formatted *WRITE* statement is split for printer files in Cobol into its two components of the encoding and the putting out of the character string.

Fault trapping on I/O is poorly specified in Cobol, and reinforces the impression of rigidity given elsewhere. The low-level manipulation of character strings (to say nothing of bit-strings) is missing, constraining the programmer to certain forms of item only, and those treated differently in input and output. The reaction of the programmer will depend on his preference for the picture method of external representation and his tolerance of the rather cumbersome method of declaration. Compare, for example the declarations

INTEGER COUNT

with format I4, of Fortran, with Cobol's

*COUNT PICTURE S*9(3) *COMPUTATIONAL*

4.4. *PL*/1

PL/1, developed by IBM and as described in the language specification (1965), contains elements from Algol, Cobol and Fortran. The I/O operations include those of Fortran with extensions, some deriving from Cobol. There are many more item types in PL/1 than in Fortran. Those that may have an external representation are listed below with examples of how they and their precisions are declared. Examples of their representation externally are also given (2.3.4L is a sterling constant).

Fixed point decimal	*DECLARE PI FIXED DECIMAL* (7, 6)	3·141593 or 2.3.4*L*
Binary fixed point	*DECLARE I BINARY FIXED* (20, 2)	101·11*B*
Decimal floating point	*DECLARE A DECIMAL FLOAT* (5)	47·36*E*4
Binary floating point	*DECLARE X BINARY FLOAT* (16)	101·110*E*-2*B*
Character string	*DECLARE CH CHARACTER* (6)	*'PAGE 5'*
Bit string	*DECLARE BB BIT* (64)	'10' *B* or (64) '0' *B*

Operations on items include checks against the declared precisions and, if necessary, conversion from one representation to another, although an implied conversion does not always give the expected result. If, for example, *CH* is a character string with value "4·31" and *A* is a fixed point variable with precision (4, 0) then the statement

$A = CH$;

causes the character string *CH* to be converted to a fixed point number with no decimal places before assignment to *A*, so that *A* is set to 4·0. This statement thus implies the decode operation. If the character string *CH* includes a character that may not be part of a fixed point number, a conversion (or decode) error is diagnosed. The character string facility includes arrays of type character as proposed in Section 5.1. The declaration

DECLARE BCD(80) *CHARACTER*(1)

declares *BCD* as an array of character strings each of length one. The *I*th character may be referred to as *BCD*(*I*). The *additional* declaration:

DECLARE X CHARACTER(80) *DEFINED BCD*

declares *X* as a character string of length 80 superimposed on array *BCD*. Thus *X* refers to the complete set of 80 characters and *BCD* allows access to individual characters. The *SUBSTR* pseudo-function allows access to sequences of characters making up a character string. For example, *SUBSTR*(*X*, 11, 10) refers to the 10 characters from *X*(11) to *X*(20).

A mixture of types may be stored in the same array using the pseudo-function *UNSPEC* as well as by using the *DEFINED* attribute shown above. *UNSPEC* suppresses the normal conversion of types that takes place during the evaluation of an expression and in value assignment. However, there is no guarantee that the assignment will work in the same way on all computers.

The I/O operations of PL/1 may be divided into two classes, according as they reference record-oriented or character streams. Record-oriented streams consist of a series of records each to be read as an entity. For example the statement:

READ FILE (file name) *INTO* (structure name);

reads the next record from the file specified into the area of store associated with the structure specified. That is, the list of items in the record is as declared in the definition of the specified structure. The data structures in PL/1 may be regarded as defining the rules by which values in the structure may be accessed. It is possible to define a structure without associating the access rule with a particular sequence of stores. Thus a record may be read into a buffer and a predefined structure then associated with this buffer.

The operations referencing character streams assume that the I/O stream is a continuous sequence of characters without characters such as newline. For example, if the input stream originated from a card reader and an input statement reads items from the first 30 columns of a card the next statement will begin from column 31 of this card rather than column 1 of the next card as in Fortran. The operations of buffer in and decode may be implied by one input statement and buffer out and encode by one output statement. However, they may also be separated. For example the instructions

GET FILE (*IN*) *EDIT* (*CH*) (*A*(80));
GET STRING (*CH*) *EDIT* (*B*1, *B*2, *B*3) (*R*(*FMT*));

read 80 characters into the character string *CH* and apply the format labelled *FMT* to decode *CH* to form the items *B*1, *B*2 and *B*3. The character string is not disturbed and it is therefore available for further decoding using other formats. Notice that these statements simulate the normal Fortran methods of input since 80 characters are read at a time and if the first use of this *GET FILE* instruction begins at the beginning of a card subsequent uses will also.

I/O statements referencing character streams are of three kinds. List-directed transmission contains a list of items to be transmitted and a standard format decided upon by the local operating system is used. For input this is usually a form of free-field representation in which blanks between numbers are ignored and the separator is a comma. Data-directed I/O corresponds closely to the Namelist feature of Fortran. Edit-directed I/O is controlled by a format statement and examples have been given above. In addition a picture form of I/O similar to that of Cobol is available.

It is an important feature of PL/1 that the I/O operations apply to files which are not associated with a particular peripheral during the program execution. Peripheral independence is thus assumed by the language and it is an operating system function to associate a file with a peripheral. Thus user's input and output files may be re-processed during execution. Output remaining in an output file at the end of the execution will be sent to the specified peripheral by the operating system.

PL/1 external types and specification of external position are basically symmetric for input and output and the asymmetry noted for Fortran due to the use on output of the first column for printer control has been abandoned. Instead there are elements that may be included in a format list for this purpose. *PAGE* requests a new page, *SKIP*(w) causes w lines to be skipped and the column position to be set to one, *LINE*(w) causes lines to be skipped until line w and the column position to be set to one, and *COLUMN*(w) causes the next item to be encoded beginning at column w, although this may not be used for back-spacing.

A valuable feature is the ability to declare page and line size for a print file at the time the file is opened. Whether or not a file with the print attribute can be accepted on a peripheral such as a disc area is a decision for the operating system. The asymmetry noted for Fortran between input and output caused by accepting input presentations of *E*, *F* and *D* items that are different from the format used on output is present in PL/1 also. That is, input is more flexible in representing an item than output. However, it is possible to output items and later re-input them using the same format list. The reverse process of outputting items with the same format list as was used on input may not be possible. A further source of flexibility comes from the ability to include parameters in a format list. For example the format element

$(N)F(I, J)$

specifies that N numbers are to be input with a field width of I and with J decimal places.

Finally we must mention the *ON* statement. This enables the programmer to declare what action is to be taken if particular errors are detected. For example, the statement:

ON ENDFILE (filename) *GO TO RECA*;

specifies that if an attempt is made to read beyond the end of the file named then control is to transfer to the statement labelled *RECA*. Other actions may be specified for other error conditions including an undefined file referenced, and end of page. Different actions for the same error may be specified in different parts of the program.

5. Proposed Fortran Extensions

We consider here how Fortran may be extended to provide the facilities discussed in Section 3. It would be possible to define and implement a package of subroutines for I/O but universal agreement on the specification of the package would be required for this to be a solution. We are aware of several such packages already. However, we have opted to make tentative proposals for extensions to the Fortran language. The proposals fall into two classes, namely the definition of further basic item types in the language, and further I/O instructions to separate input and output into their constituent stages and provide good error recovery procedures.

5.1. *Type Character*

It should be possible to declare an array of characters using a declaration such as

CHARACTER CH(80)

and to refer to individual characters in the obvious way, for example, as *CH*(7) and *CH*(*I*). The basic operation of converting from a character to an integer could be made available as a system function such as

I = *IFROMC*(*CH*(12))

This could refer to a user-defined character code if declared or to the particular computer's code otherwise. The reverse operation of creating a character from an integer could be provided similarly, for example, as

CH(1) = *CFROMI*(*I*)

and would return a fault character if the value of *I* is out of range. Alternatively mixed-mode assignment rules could be defined for the automatic conversion of items in assignment statements. The vital operations of packing characters into a single word and unpacking words to form characters would best be provided by system subroutines. For example,

1. *SUBROUTINE* PACK (*C*, *NC*, *W*)
   Would pack *NC* characters in array *C* to form words in array *W*.
   SUBROUTINE UNPACK (*C*, *NC*, *W*)

Would unpack *NC* characters from array *W* to form array *C*. Implementation of these subroutines would require some means by which the user could declare the number of bits per character. The number of characters packed into one word is machine-dependent and it should be available at the Fortran level.

It is very important to be able to involve individual characters in comparison (*IF*) statements. For example,

IF(*CH*(4) .*EQ*. *X*1) *GO TO* 6

would mean that control would transfer to the statement labelled 6 if the character in *CH*(4) was equal to the character in scalar *X*1. If *X*1 is also declared as type character the comparison is straightforward. However, if *X*1 is not type character a conversion is implied which would produce the first character in *X*1. The scalar *X*1 could be set using a *DATA* initialization statement as words are set now. That is, for example,

*DATA X*1/1*H*,/

would be used to set *X*1 to contain a comma. Similarly if we set *X*1 by

*DATA X*1/4*HABCD*/

the *IF* statement above would compare *CH*(4) with the character *A*.

5.2. *Type Bit*

The input and processing of binary data (as discussed in Section 2.5) would be greatly simplified if it were possible to define an array of bits. The obvious declaration would be

BIT B(120)

with reference to individual bits using the obvious subscript notation; for example *B*(5) or *B*(*J*). Conversion to and from integers and characters could be achieved as for characters, using system functions, and an agreed conversion between types bit and logical would be necessary. Conversions between integer and bit would involve values 0 and 1 only, conversions to and from characters would involve characters 0 and 1 only, and conversion to and from type logical would equate a one bit to "true" and a zero bit to "false".

In addition to these conversion operations packing and unpacking operations could be made available through system functions. Conversions between type bit and types integer, character, and a word representation, also necessary, would assume that the bits are ordered from left to right in their storage locations. The number of bits per word is machine-dependent and it should be available through a system function.

5.3. *Proposed I/O Extensions*

The extensions cover the following operations:

1. The declaration of buffers for I/O operations.
2. The transfer of records between files and buffers.
3. The decoding and encoding of items.
4. Output format functions.
5. Functions on buffer pointers.
6. Item and record status functions.

5.3.1. *Declaration of buffers*

Syntax:

⟨label⟩ *BUFFER* (⟨stream number⟩, ⟨address⟩, ⟨length⟩)

Semantics:

BUFFER is a declarative statement associating an array in core whose first element is "address" and whose length is "length" with a stream number and a label. The stream number is associated with a file through the operating system and the label identifies the buffer and stream in future references. The length is in units of store depending on the type of the address identifier.

5.3.2. *Transfer of records between files and buffers*

Syntax:

INPUT ⟨buffer label⟩
OUTPUT ⟨buffer label⟩

Semantics:

INPUT gets the next record to the buffer whose label is given from the file associated with that buffer. It resets the associated pointer to 1 (see Section 5.3.5) and sets a status word referenceable by functions described in Section 5.3.6.

OUTPUT puts the next record from the buffer to the file, sets the pointer to 1, "blanks" the buffer and sets a status word similarly. Some means of suppressing the blanking of the buffer, however, is required.

Note: the *ASSIGN* facility could be extended to allow the buffer label to be replaced by an identifier.

5.3.3. *The decoding and encoding of items*

The general form is:

$\left\{\begin{matrix}\text{DECODE}\\\text{ENCODE}\end{matrix}\right\}$ (⟨identifier⟩) ⟨er⟩, ⟨field width⟩, ⟨ndp⟩

The statements refer to the buffers last declared by *INPUT* and *OUTPUT* respectively and *er* denotes external representation and will have code letters as values. For scientific work the following will usually suffice:

B	Bit string
C	Character string
D	Double-length real
E	*E*-format real
F	*F*-format real
I	Integer
L	Logical ("true" or "false")
<u>*R*</u>	Real embracing *D, E, F* or *I*

(underline denotes default setting)

field width	If positive this denotes fixed field of that width. If zero this denotes do nothing. If blank or negative it denotes free-field. Its format is an integer expression.
ndp	The number of decimal places. Integer expression. Can be omitted if irrelevant (character string) or unset (unspecified real number). Default value zero.

The identifier defines the internal item. Its internal representation may or may not be of the same type as the external representation. When not, the same operations as in mixed-mode assignment take place.

Examples

Let *A* be *real* and *I* an integer, then

DECODE (*A*) *E*, 12, 4	As in Fortran *E*12·4
DECODE (*A*) *R*, , 1	Real number, free format, 1 d.p.
DECODE (*A*) *C*, 4	Fill with 4 characters
DECODE (*A*) *B*, 10	Binary string of length 10
DECODE (*A*) *L*, 1	*A* = 1·0 if next character is *T*, = 0·0 if next character is *F* else fault
DECODE (*A*) *L*, 5	*A* = 1·0 if next 5 characters contain the sequence *TRUE* = 0·0 if *FALSE* else fault
DECODE (*I*) *I*, 6	As in Fortran *I*6
DECODE (*I*) *C*, 4	Fill with 4 characters
DECODE (*I*) *R*, 8, 2	Truncate real number
DECODE (*I*) *L*, 1 / *DECODE* (*I*) *L*, 5	As for examples on *A* above but integer setting

For *ENCODE* er-code *R* is interpreted as *E*. If the field width is omitted it is taken as the smallest number of characters needed to output the item.

The detection of faults in the encode/decode process is dealt with in Section 5.3.6.

5.3.4. *Output format functions*

There are three:

SPACE (*n*)
NEWLINE (*n*)
NEWPAGE (*n*)

The one parameter is an integer expression for the number required. They refer to the buffer defined in the last *OUTPUT* statement. "Space" has the same effect as the "skip" function in Section 5 if the buffer concerned has not been written into. "Newline (3)" encodes 3 newline characters into the current output buffer and is equivalent to

OUTPUT (*n*)
OUTPUT (*n*)
OUTPUT (*n*)

i.e. to outputting the next record, followed by 2 blank records, from the current output buffer. "Newpage ( )" encodes newpage characters or their equivalent into the current output stream.

5.3.5. *Functions on buffer pointers*

Each declared buffer has an implicit pointer, which at any time, has as its value the position of the next character to be used in an encode/decode statement. Two functions are essential, to get and set the pointer value, and a third, to advance the pointer value, is convenient. The parameters are

GET POINTER (⟨buffer label⟩)
SET POINTER (⟨buffer label⟩, ⟨value⟩)
ADVANCE POINTER (⟨buffer label⟩, ⟨value⟩)

If a buffer is of length *n*, then the pointer will be set to 1 or *n* if the calculated value falls below or above these extremes respectively and the record status word will be set accordingly (see Section 5.3.6.).

5.3.6. *Item and record status functions*

These are provided to enable the programmer to retain control when things go wrong. They return an integer value and have a buffer label as parameter. For item status the following values could be set.

0	Last item decoded/encoded successfully
1	Field out of range (i.e. extends past end of record)
2	Field width too small (encode only)
3	Invalid character (decode only)

For record status values could be set

0	Last record transmitted successfully
1	Attempt to set pointer out of range
2	Record too long for buffer
3	Attempt to read past end-of-file

In addition the following information should be obtainable after a decode or encode instruction:

(i) The number of leading blank characters.
(ii) The number of non-blank characters in the item.

REFERENCES

COOPER, B. E. (1968). Basic subroutine for the input of numbers, words, and special characters. *Computer J.*, **2**, 157–159.

IBM (1965). PL/1 Language Specification File No S360–29, Form C28–6571–4.

IFIP Working Group 2.1 (1964). Report on input–output procedures for Algol 60. *Comm. Ass. Comp. Mach.*, **7**, 628–630.

KNUTH, D. E. *et al.* (1964). A proposal for input–output conventions in Algol 60. *Comm. Ass. Comp. Mach.*, **7**, 273–283.

SINGH, G. (1968). The M.V.C. Manual, available from Information Officer, University of London Atlas Computer Service, 39 Gordon Square, London, W.C.1.

Report on the Work of the Dutch Working Party on Statistical Computing

By A. J. VAN REEKEN

"*Those prophets who really knew, were stoned one by one*"
NIC VAN ROSSUM, in *De Tijd* (March 23rd, 1968)

INTRODUCTION

AS IT is hardly interesting for you to listen to someone merely reporting on what work has been done, I thought it might be interesting to tell you first what work could be done, and then to summarize how we thought it should be done, and what we have achieved.

This seems justifiable, because although both working parties, British and Dutch, are concerned with statistical computing, there is a considerable difference in point of view and mode of operation.

The Dutch Working Party started on the basis of the following observations.

1. A TECHNICAL TOOL OF STATISTICS

The computer is used as a computing slave, a producer of tables and writer first class (be warned: by no means a first class writer!).

That is, once the technique to be used has been specified, then, given the data and necessary parameters for options, the computer will apply the technique and produce the results. It does not check the data for consistency with the applied technique, and, consequently, automatic application of an alternative technique (for example Wilcoxon's instead of Student's t-test) is out of the question.

It happened quite often to me in my previous job that statistical advice was asked only after the data had been collected. Most of the time one could do no more than try to minimize the damage. That meant taking care that no unjustifiable inferences were drawn, or refusing the project. The researcher involved often learnt by this procedure and came earlier next time. In recent years, mainly because I am somewhat closer to the computer than to statistics nowadays, it appears to me that where standard programs for a computer are available with good manuals, the amount of advice requested is decreasing.

However, if you conclude from this that, compared to seven years ago, many more researchers know about statistical "may's" and "may-not's", you are mistaken. In the Netherlands, I know a research worker who computed the coefficient of correlation for variables "civil state" and "religion". And this is not an isolated example.