Tree driven data input and its validation

M. I. Padgett

Department of Civil Engineering, University of Nottingham, University Park, Nottingham, NG7 2RD

A generalised technique for the input of forms into a computer system, and its implementation in ALGOL is described. The format for any section of a form is parameterised. Each type of form has an associated input format tree which holds the various sets of section parameters. This input format tree is used to decode the data belonging to a form into a data tree.

(Received September 1972)

The work described in this paper was one part of a larger system known as ADMIN (Automatic Design of Management Information Networks). Although ADMIN was not fully implemented and work on it has now terminated, the part described below was fully implemented and tested. Furthermore, it can be described on a standalone basis. ADMIN, in terms of business systems, was aimed at promoting the collection of the correct data in a sufficiently short timescale for economic analysis to be feasible. To this end the system was intended to provide a facility for the construction and interrogation of models which consist of highly interrelated sets of items (i.e. networks). The method of building these models was meant to be by the input of Standard Forms to the computer.

The main difficulty in designing an inputter for these Standard Forms is that it is not known at the outset what type of forms are required to gather the data for a particular management information system. The choice is either to have purpose built inputters for each type of Standard Form or to have one generalised inputter capable of handling all types of forms; the implications of adopting the latter being that it will be highly parameterised. Because of the desire to have high flexibility and speed in allowing new forms into the model, the generalised inputter is preferable and is the one described in this paper.

The paper has three main sections. The first indicates in an informal manner how the structure of a form can be analysed. The second formulises this analysis by parameterising each section of a form. In the third section an attempt is made to give an insight into the computer techniques used in the implementation.

Having gathered the data on one form it is often transcribed onto a suitable form for data preparation. This method is obviously open to clerical error. The prime design aim of the inputter was to facilitate the completion of Standard Forms with a minimum regard for the need of the computer to have 'punctuation' between sections. In view of this the inputter allows a very free format and forms are suitable for immediate data preparation after the addition of a small number of section delimiters.

The inputter was implemented on ICL 1900 series computers. The bulk of the coding was in ALGOL which was enhanced by some fifty or more primitive subroutines written in PLAN. These primitives enabled structure manipulation to take place and were available to all ADMIN subsystems.

This paper is *not* intended to be a full specification for the inputter. Space alone forbids this and in addition it would tend to cloud the main ideas. The aim is to give the reader a feeling for the problem and how it has been solved. Every effort has been made to ensure that the rules and definitions given are consistent but no formal attempt has been made to be 'necessary and sufficient'. Appendix 1 gives an explanation of the technical terms used. It might be found helpful to read it at this juncture, and in this manner it is hoped to establish a common vocabulary with the reader for *this* paper.

Informal discussion of the main concepts

The basic aim of the inputter is to decode the data on a form into a data tree and perform a limited amount of validation on the data. This aim is achieved by the use of an input format tree of which there is one for each type of form. This data tree is then passed to other ADMIN subsystems for further semantic checks before items from it are placed in a database. First some of the basic syntax of a form is analysed and then some of the facilities available to the user are described.

In general a form can be divided into *sections*, either by physical lines or a numbering system, or a combination of both. The two methods of division are exactly the same from a logical point of view. The idea of a section is now developed. A section is said to be complex if its contains subsections and primitive if it contains no subsections. A primitive section contains a string of characters or an integer (in effect a string of characters which are interpreted).

A convenient method of holding the structural information implied by dividing the form into sections is to represent it by a tree, each section of the form corresponding to a node of the tree. The tips of the tree corresponding to primitive sections and other nodes corresponding to complex sections. **Fig. 1** shows how the structure of a form can be mapped onto a tree structure.

The reasons for using a tree to represent the structural information must be understood in the wider context of ADMIN where the idea of a network is basic. A tree can be considered as a special case of a network and is thus an obvious structure with which to identify the format of a form. Further to this, many primitive procedures for the manipulation of trees are available to all the ADMIN subprojects of which this is one.

In Fig. 1 one primitive has been indicated as having a variable number of characters. In such a case a single character, referred to as a *delimiter*, is added to the string of characters, so that when decoding of the data takes place a definite symbol indicates the end of the primitive. The choice of character which is to act as a delimiter is free as long as it obeys the rules which are defined later. Often a delimiter will occur naturally on a form as for example, in a list of names separated by commas. The comma acts as a delimiter but it is *not* part of the data. If the primitive is held in a fixed length field as for example:

```
⌊ F L O W ⌋
```

then it can be argued that there is no need for a delimiter, since six characters are taken to be the primitive, and leading and trailing blanks are removed. However, fixed format on forms is severely restrictive and especially inappropriate when the format of forms might change quickly in the initial design stages of any system. Delimiters, as will be seen later, have a much more general use than the one outlined above.

Frequently on a form a section, either primitive or complex, will be left blank, as it is found to be nonapplicable. Formally

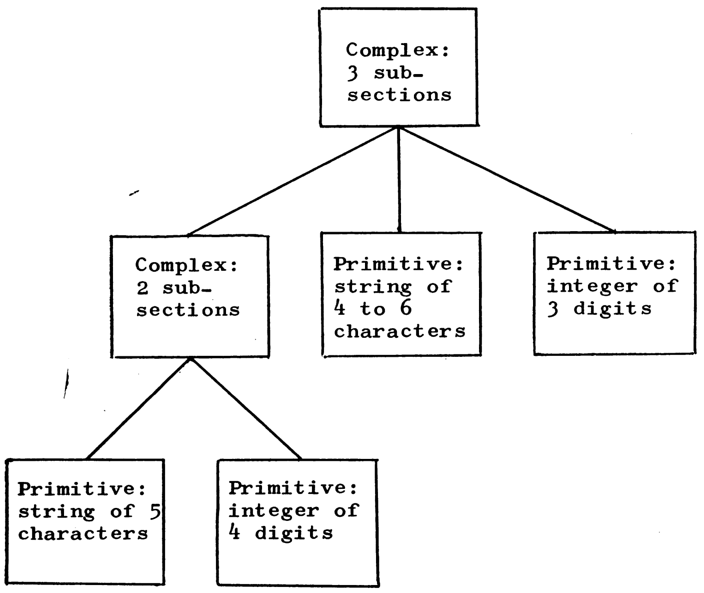

Complex: 3 sub-sections
Complex: 2 sub-sections
Primitive: string of 4 to 6 characters
Primitive: integer of 3 digits
Primitive: string of 5 characters
Primitive: integer of 4 digits

Fig. 1 Structure of a form mapped onto a tree structure

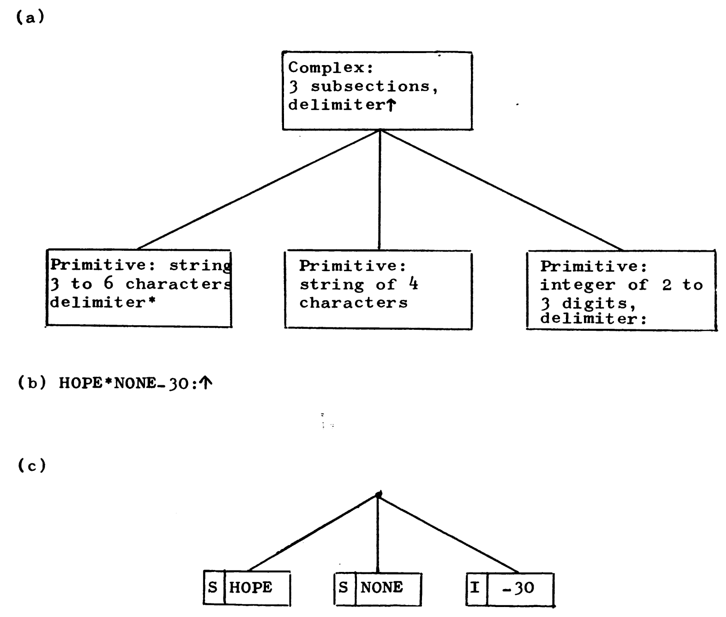

(a)
Complex: 3 subsections, delimiter↑
Primitive: string 3 to 6 characters delimiter*
Primitive: string of 4 characters
Primitive: integer of 2 to 3 digits, delimiter:
(b) HOPE*NONE-30:↑
(c)
S HOPE
S NONE
I -30

Fig. 2 (*a*) **An illustration of a simple input format tree with delimiters**
(*b*) **Specimen data**
(*c*) **Data tree that would be produced using** (*a*) **and** (*b*)
(N.B. In (*c*) **S denotes a string and I an integer)**

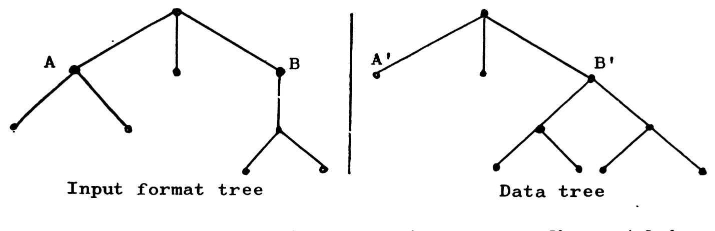


Fig. 3 (*a*) **Contraction at A′ due to section corresponding to A being null.**
(*b*) **Lateral extension at B′ due to repetition of section at B.**

such sections are said to be *null*. If the section is primitive, then one method of indicating nullness is to punch space characters. However, it is often more desirable to have a visible character which can be used to represent nullness. For example, positions not occupied by digits in a cheque might contain the character *, and this is an obvious security precaution.

If the primitive section has a delimiter, then a shorthand way of denoting nullness is simply to punch the delimiter. This is one reason why primitives containing a fixed number of characters are allocated a delimiter. In this case if the primitive is null, then a convenient way of indicating nullness is to punch only the delimiter as opposed to punching a fixed number (possibly large) of null code characters if no delimiter is allocated. The example below shows the three possible ways of indicating a primitive as being null, assuming a *nullcode* of * (say) and a delimiter : (say).

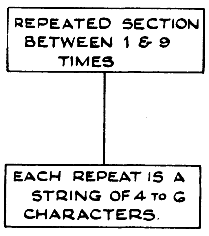
Example

□ □ □: i.e. spaces and delimiter
***: i.e. nullcode and delimiter
: i.e. delimiter only.

The shorthand used in denoting nullness of primitive sections (i.e. delimiter only) becomes essential when considering the nullness of complex sections. Each complex section is assigned a delimiter. With one optional exception (described later) this delimiter is always added to the end of any data concerning the subsections of this complex section. A simple example of the input format tree and a specimen data input is shown in **Fig. 2.**

To denote a complex section as being null it is sufficient to write its delimiter. It should be noted that the depth of the data tree is less than the corresponding part of the input format tree, if a complex section is null. **Fig. 3** shows the structure *only* of the input format tree and the data tree. The complex section corresponding to the node marked A is null and thus there is contraction at the node marked A′.

A common occurrence on forms is for a subsection format to be repeated several times. An example of this might be in forms describing the job functions of personnel where each line corresponds to one person. It would be inefficient to repeat the identical structures on the input format tree, and so the concept of a repeated section is introduced. All the repeated sections on a form might not be used in some cases, and thus the maximum and minimum number of expected data sets for this structure is specified. The example below illustrates how a section of the structure tree might look.


REPEATED SECTION BETWEEN 1 & 9 TIMES
EACH REPEAT IS A STRING OF 4 TO 6 CHARACTERS

It will be noticed there is only one son from the repeated type node. This son corresponds to the repeated section and might be complex or primitive. In Fig. 3 the node B corresponds to a repeated section. It will be noted there is a lateral extension of the structure at node B′ on the data tree.

Sometimes data in one or more subsections of a *repeated* complex section will overflow into the same subsections of the next repeated complex section. Typically this occurs in repeated sections which are lines divided into subsections. **Fig. 4**(*a*) shows a form with four subsections on each repeated line for name, job description, salary and age respectively (say). No delimiters are shown but they are, of course, necessary. The job description has overflowed. Initially the inputter decodes it as two independent repeated subsections as shown in **Fig. 4**(*b*). *Resequencing* then takes place as shown in **Fig. 4**(*c*), so that the result is one logical section.

A method has been evolved to indicate on the input format tree when resequencing is allowed and what 'signal' should trigger it.

A limited amount of what is best described as syntactic and semantic *validation* is carried out on primitive sections. A primitive is checked to see that the number of characters it contains falls within a defined minimum and maximum.

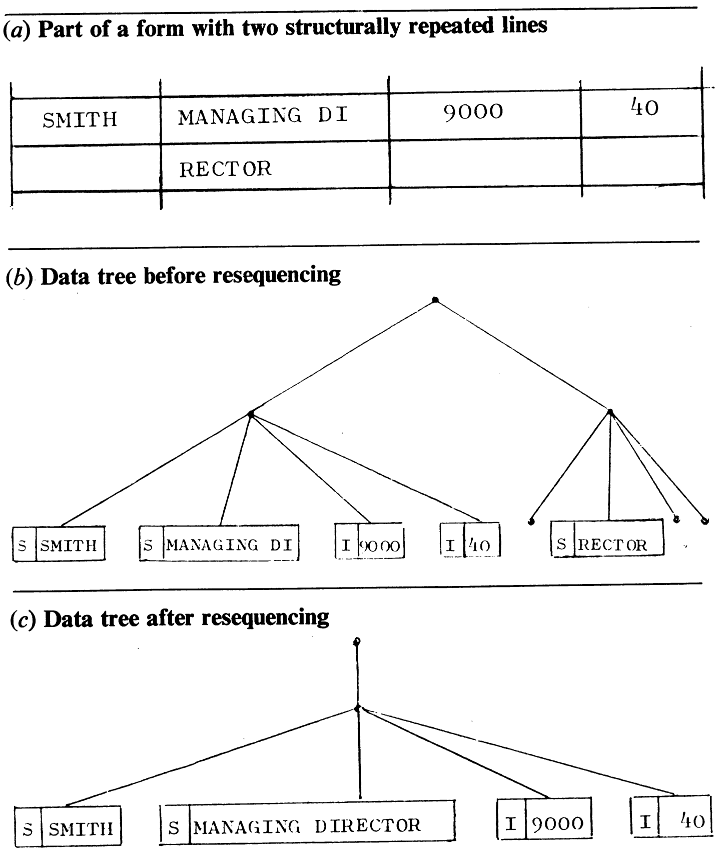

(a) Part of a form with two structurally repeated lines
SMITH
MANAGING DI
9000
40
RECTOR
(b) Data tree before resequencing
S SMITH
S MANAGING DI
I 9000
I 40
S RECTOR
(c) Data tree after resequencing
S SMITH
S MANAGING DIRECTOR
I 9000
I 40

Fig. 4 Resequencing.

Integers (i.e. converted strings) are optionally checked that they are less than a given maximum and optionally greater than a given minimum. No contents check is carried out on a string not representing an integer, but it is possibly desirable and can with little modification to the inputter be included. Other types of validation take place between sets of items and these are mentioned elsewhere. It should be noted, however, that anywhere a rule is stated or implied there is a corresponding error check.

Formal description of node parameters

The eight parameters which are used to define the syntax and validation associated with each section are now given. In the following node and section should be taken to be synonymous.

1. *TYPE*

Complex sections are either of type M or type R.

(*a*) Type M if it contains a *fixed* number of subsections, not all of which are identical. (M for many or multiple or mixed.)
(*b*) Type R if all the subsections are identical.

Primitive sections are of type S or I.

(*a*) Type S is a string.
(*b*) Type I is an integer.

2. *MAX*

(*a*) For type M or R specifies the maximum number of subsections or repeats respectively.
(*b*) For I or S specifies the maximum number of characters.

3. *MIN*

As in MAX, with minimum replacing maximum.

N.B.

1. For type M then : MAX = MIN.
2. For fixed length primitives MAX = MIN.
3. Leading and trailing spaces are included in MAX, MIN conditions. But will be removed during processing.
4. The sign character if present is included in MAX, MIN conditions for integers.

4. *DELIMITER*

Delimiters are used to terminate or denote the nullness of complex or primitive sections. A delimiter must be defined for all sections except for a primitive section with a fixed number of characters (i.e. MAX = MIN) where its definition and subsequent use is optional. Delimiters cannot be defined in isolation nor can they be used on particular data forms without corresponding to the rules given below. However, first the concept of an *effective delimiter* must be established.

Any section which is the last subsection (i.e. youngest son on the input format tree) of another section can, if a delimiter is required, use the delimiter of main section. This concession is useful, since it cuts down on the number of non-data characters on a form. The idea can be extended so that one delimiter can act for several nested subsections each of which is a last subsection. **Fig. 5** shows a skeleton tree with delimiters at the side of each node. A legal delimiter sequence for nodes 8, 7, 1 would be

$$* : \uparrow$$

but in fact ↑ can be used to the same effect. Rules governing the definition and use of delimiters are now given.

Rule D1

No delimiter on the input format tree for a form can appear in a data item for that form unless preceded by the transparency character (@ in the current implementation).

Rule D2

The transparency character @ and the effective end of line marker ← (described later) are not allowed as delimiters.

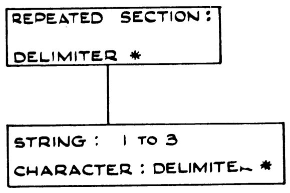
Rule D3

The shorthand technique of using effective delimiters is not allowed when father and last son have the same associated delimiter. If the father delimiter is present, then should the presence of the son be indicated *it* must also be accompanied by *its* delimiter. Three legal sequences of delimiter following any data corresponding to node 10 in Fig. 5 are:

1. **:
2. *: (: acts as an effective delimiter for node 8)
3. : (: acts as an effective delimiter for both nodes 10 and 8)

Rule D4

The effective delimiter for a set of repeated sections must not be the same as that for the repeated section. For example: if the structure tree is as below:



then ABC ** is ambiguous since the second * can either indicate a null string or the end of the repeated set of strings.

Rule D5

If a complex section is to be denoted as being null by using only its delimiter, then the following condition must hold.

Condition

No node on the yo-yo path to a primitive node shall have the same delimiter.

If this condition fails, then the nullness of all the sons of the complex section must be indicated, and *Rule D5 applies to all of these sons.*

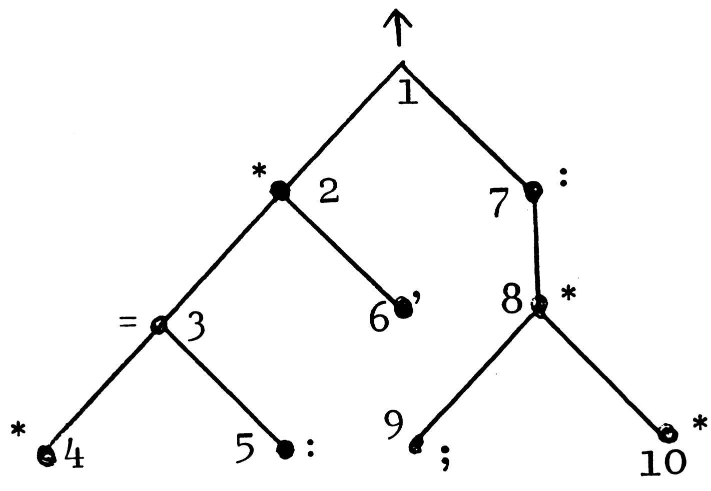


Fig. 5 Skeleton input format tree with delimiters.

As an example of the use of Rule D5, consider the structure tree in Fig. 5. If the section corresponding to node 2 is null then =, * is required and not just *.
N.B. This is the shortest way. A further legal way is * : =, *. Of course if the primitives have null codes these can be optionally inserted.

The above rules might appear unwieldy, but they are given to allow for input format trees of any size. In practice it is common for them to have about a dozen nodes, and the rules then become very simple to operate.

5. *NULLCODE*
This parameter is applicable to primitive sections only (i.e. type S or I). The character may be inserted between MIN and MAX number of times to indicate that the primitive is null.

6. *MINVAL*
Applies to nodes of type I and specifies the minimum allowable value of the integer.

7. *MAXVAL*
Applies to nodes of type I and specifies the maximum allowable value of the integer.

8. *RESQ*
Applies only to nodes of type R. Resequencing is allowed to take place between data sets whose structures are identical. The structure of these data sets is defined once and hangs from a node of type R. For two data sets the nullness of a particular subsection in the second set will trigger resequencing. RESQ is set non-null if resequencing is desired, and its value indicates which son in the data set is to trigger resequencing.

Two rules are necessary:

Rule Q1
If either data set is completely null in all its sections then no resequencing takes place.

Rule Q2
At least one of any two corresponding integers must be null.

Example: $\frac{|\ 42\ |}{|\ 37\ |}$

will *not* be resequenced as 4237.
Note:
The usual removal of leading and trailing spaces is not done until all resequencing has finished. Thus intermediate spaces are preserved.

The decoding technique
Assuming that the data has been read into the input buffer† and the requisite input format tree has been obtained the general principles needed to obtain the corresponding data tree are now given. **Fig. 6** is used as an example and is referred to in the text. The input format tree and the data have been chosen so as to illustrate many of the ideas mentioned previously. In this respect they are not completely typical of everyday use. The contents of the form are self-evident. Double underlining represents the pre-printed section of the form which is not input as data. Error detection and recovery are discussed in a separate section but it must be realised that these processes are continuing simultaneously with the formation of the data tree.

1. *Recursion*
The main decoding procedure is recursive. Some justification for this statement is now attempted.

1. Tree structures are essentially recursive, i.e. each subtree taken in isolation is a tree in its own right.
2. The decoding of a complex section is not completed until its last subsection has been decoded. Thus at any one time there are several sets of the eight-parameters associated with each section in operation. They correspond to different section levels. These parameters have to be stored and referred to at various stages. Thus an obvious technique is to use a recursive procedure where these parameters can automatically be stored and destroyed after use.
3. There is a one to one correspondence between the depth in the recursion and depth in the input format tree. Furthermore there is, in effect, a one to one correspondence between the *actual* calls of the recursive procedure and the *actual* number of nodes that are examined during decoding. The reason for the words 'in effect' is that for repeated sections some nodes will be examined as many times as there are data sets corresponding to the repeated section.

The onus for going deeper into the recursion rests with the input format tree whilst exist from recursion depends on the presence or lack of data in the input buffer. Which of the two predominates is now discussed in Sections 3 and 4 below. First, however, the technique used to form the data tree is described.

2. *Method of forming the data tree*
If a section is found to exist, after analysing the input buffer, then a data tree corresponding to that section is passed out of the recursion to the level above. This data tree will be degenerate, i.e. consist of only one node, in the case of primitive sections and in the case of complex sections that are null. At the level above this data tree now becomes the youngest subtree on the data tree which is being formed at that level. If it is the first subtree at the level above, then it will entail the creation of a head node from which it can be hung. **Fig. 7** shows how the subtree of Fig. 6(*c*) containing the data HERBERT, CLERK, 1200 has its final section added.

3. *Principle of maximum recursion*
The recursive procedure starts operating by examining the data at the head node of the input format tree. (This can be considered to correspond with a complex section for all practical purposes, since the case of a form consisting of a single string or integer is unrealistic.) For any complex section, if data is present corresponding to the first primitive section contained within it, then assuming no errors the current position pointer

†Conceptually the input buffer consists of a linear string of the data characters and a current position pointer able to move in both directions. Each card has an end of line marker (← in the current implementation) and only data up to but not including ← is appended to the buffer.

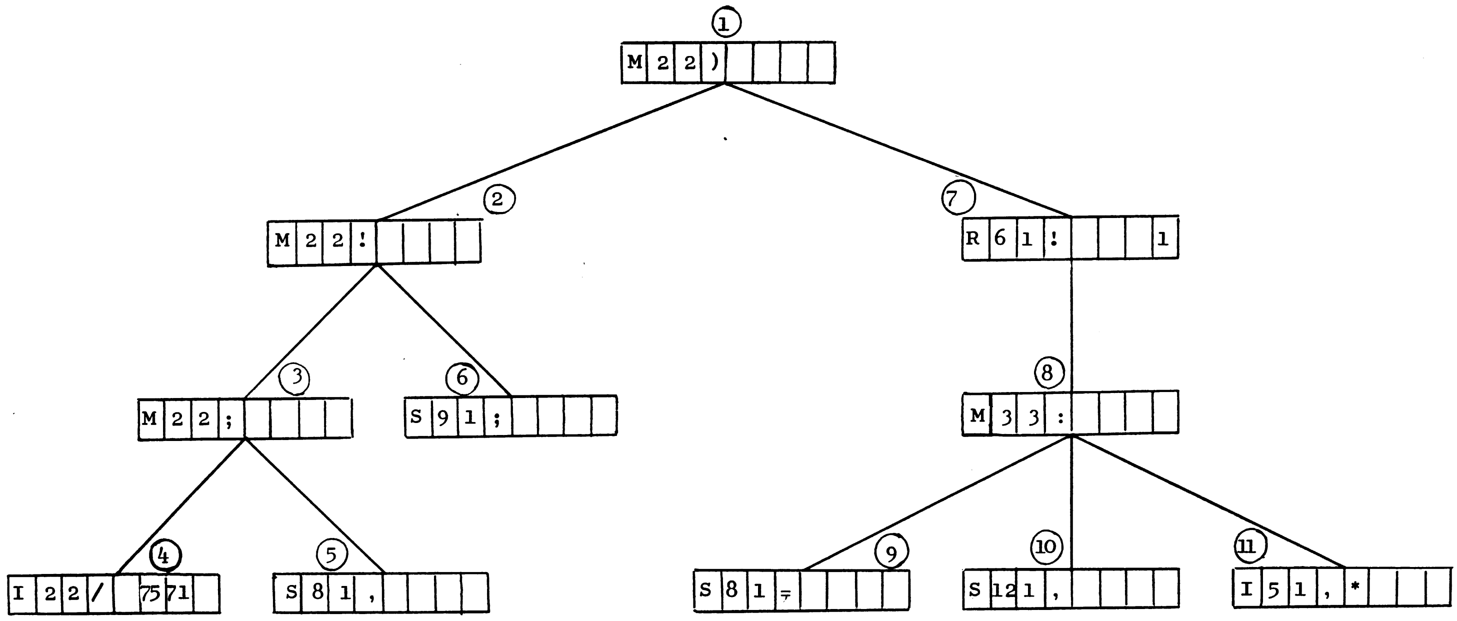
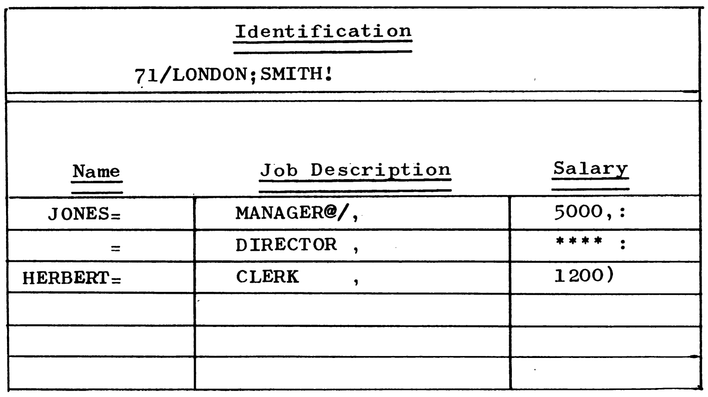


Fig. 6 (*a*) An input format tree with parameters from left to right at each node



Identification

71/LONDON;SMITH!

Name	Job Description	Salary
JONES=	MANAGER@/,	5000, :
=	DIRECTOR ,	**** :
HERBERT=	CLERK ,	1200)

Fig. 6 (*b*) A simple standard form whose input format tree is shown in Fig. 6(*a*)

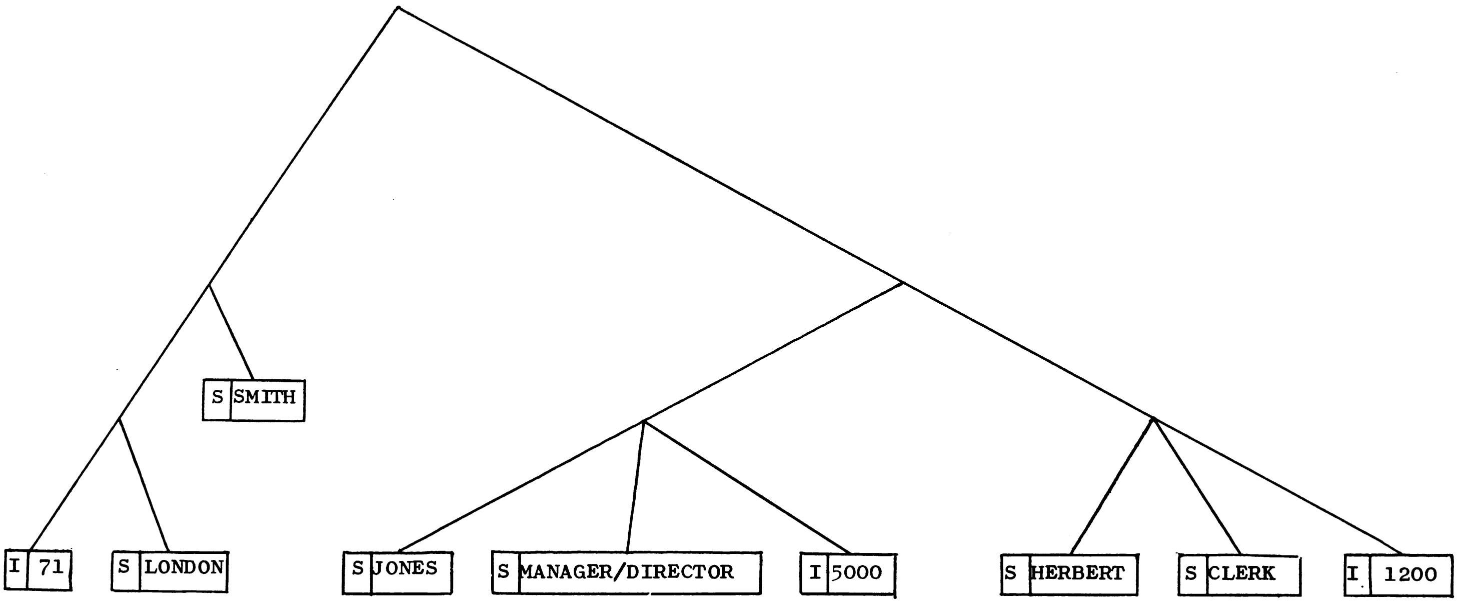


Fig. 6 (*c*) Data tree produced using (*a*) and (*b*)

of the input buffer will be pointing to the first character of that section. Accordingly recursion, and correspondingly deeper penetration of the input format tree, automatically takes place, following a yo-yo path until a primitive node is reached. It should be emphasised that at no stage during this recursion is the input data buffer examined. For example, in Fig. 6(*a*), recursion takes place from node 1 until node 4 is reached.

However, during the recursion the delimiter associated with each node is noted and placed on a stack, known as the *current delimiters stack*. It will be seen later how this stack is used for

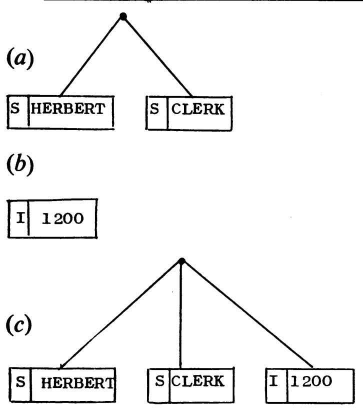


Fig. 7 Method of forming data tree
(*a*) Partially completed data tree at level *n* (say) of recursion.
(*b*) Subtree (degenerate) at level $n + 1$ of recursion
(*c*) Resultant data tree at level *n* after addition of subtree from level $n + 1$

decoding and error detection purposes. Thus at node 4, for example, the stack will contain the delimiters /;!).

By the above process an alignment of the input format tree and the input data buffer, which is likely to be correct, is achieved. The checking and correction of this, if necessary, is described below.

4. *Criteria for exit from recursion*
Depending on whether a node is associated with a primitive or complex section there are different criteria for exit from the recursion. Always, though, if a delimiter has been added to the current delimiters stack on entry it is removed on exit. The two cases are now discussed.

For *primitive sections* characters are copied from the input buffer and assigned to a section until the MAX number is reached, or a current delimiter is found. Implicit in this copying process is the moving of the current position pointer in the input buffer. If the first character inspected is a current delimiter, but not associated with the current level of recursion, then the primitive is deemed not to be present, i.e. empty and exit takes place. Note that in this case no data tree will be passed out of the recursion. Otherwise a single node data tree is passed out which either has no data, a string or an integer in its data stack.

For *complex sections* the possibility of exit is only considered after returning from the recursive level associated with the first subsection, as is required by the principle of maximum recursion. If this first subsection is empty then exit must take place. But first, the input buffer is examined to see if the character pointed to by the current position pointer is an effective delimiter for the complex section, and if this is the case a degenerate data tree with no data in the stack is passed out. Otherwise the section is empty and no tree is passed out. Should the first subsection be non-empty then a search is made for all the remaining subsections that are defined on the input format tree before exit is allowed.

Errors
After forming the data tree the most important function of the decoding process is to detect errors, note them, and if possible continue to decode. It must be stressed that recovery after an error is at best an educated guess and that it makes the decoding process considerably more complex.

As errors are found they are recorded on an error tree. The tree consists of a head node with each son of the head node corresponding to an error. In the data stack of each 'error node' are integers which parameterise the error. This error tree is then suitably presented at the end of decoding.

Some errors are localised, as for example a primitive section containing too few characters, and need no specific recovery action. Others are more serious and require a realignment of the input data buffer and the input format tree. Always it is the current position pointer of the input buffer that is moved as opposed to changing the node being analysed on the input format tree. This ensures that no looping occurs in decoding, since no back-tracking can take place on the input format tree. To elaborate, decoding ends if exit is made from the recursive level corresponding with the head node of the input format tree, or if the end of the input data buffer is reached. In normal circumstances these two should coincide. The two occasions on which the current position pointer is moved are described below.

The first occasion is concerned with an 'overshoot' situation and the possible need for 'back-tracking' in the input data buffer. A decision as to whether to back track depends on the contents of the *found delimiters stack* which is built up during the decoding process in the following manner: when decoding primitives any character found which is a delimiter, but not a current delimiter, is either deemed to have been punched spuriously, or to indicate that the input format tree is possibly out of alignment with the input data buffer. In consequence the delimiter and its position in the input data buffer are entered as an ordered pair on the stack of found delimiters. This stack is, of course, empty at the start of decoding.

If a delimiter is defined for a section, then there is an overshoot check, subject to certain conditions given below. Before this, however, the method of the check is discussed. First, all ordered pairs, that were added to the found delimiters stack when the current position pointer of the input data buffer was more than some threshold value (typically 80, i.e. one card) from where it is now, are removed. This housekeeping is designed to minimise unreliable back-tracking. Second, starting with the 'eldest', each ordered pair is examined to see if its delimiter agrees with the section delimiter. If it does, then the current position pointer of the input data buffer is moved back to the position indicated by the second element of the ordered pair, and the found delimiters stack is emptied. Otherwise the next ordered pair is examined until there are no more left.

For primitive sections the overshoot check is applied immediately on entry to the corresponding level of recursion. For complex sections, to conform with the principle of maximum recursion, the check is applied only after returning from the recursion associated with the first subsection, and then only if that subsection is empty.

The second occasion on which the current position pointer of the input data buffer is moved occurs at the end of the recursive procedure. Here the concern is that there should be no 'undershoot' by the pointer. The check is always applied to sections which have been found to be non-empty and have a defined delimiter. As an example, if a primitive has a maximum of 6 characters delimited by a colon then

SALESMAN:

will cause an error. The pointer will be stopped on the second A and thus as a corrective measure it is moved to the colon, i.e. the characters AN are ignored. Generalising, the pointer is always moved, if it is necessary, to point to the next delimiter in the buffer.

Conclusions
The inputter has relatively large overheads in the form of PLAN structure manipulation primitives, ALGOL library procedures and data storage areas. In all these come to approximately 20K words of storage as opposed to approximately 6K for the inputter itself. It was only when these overheads were

shared by other ADMIN subsystems that they became acceptable.

During the initial testing stages of the inputter over fifty different types of form were input and all the facilities mentioned were found to be useful. Typically a form with about one hundred non-null primitive sections took about one minute to decode. This time is acceptable, since for the first attempt at defining and coding the inputter, correct logic and speed of implementation were considered to be more important than efficiency. A logical system can usually be made efficient, but not necessarily vice versa.

Probably the most important spin-off from the above work is the way in which delimiters were employed. In the field of lexical analysis there would appear to be the possibility of using similar techniques. Finally it should be emphasised that the inputter was designed for large systems where quite possibly thousands of forms are involved. Its flexibility can only be justified in such an environment.

I would like to express my grateful thanks to all my former colleagues for their help, encouragement and patience. In particular, J. Thomas, who suggested the use of a tree structure technique and some of the parameters for the inputter.

Appendix 1

The purpose of this section is to define some of the terms used in this paper. No claim is made for the universality of the definitions. **Fig. A1** shows a tree structure with the nodes numbered and these are the ones referred to in definitions.

The *head* node is number 1.
Tip nodes are 3, 4, 5, 7, 9 and 10.
Nodes 2, 5 and 6 are *sons* of node 1.
Node 8 is the *father* of nodes 9 and 10.
Node 5 is the *elder brother* (or major) of node 6 and the *younger brother* (or minor) of node 2.

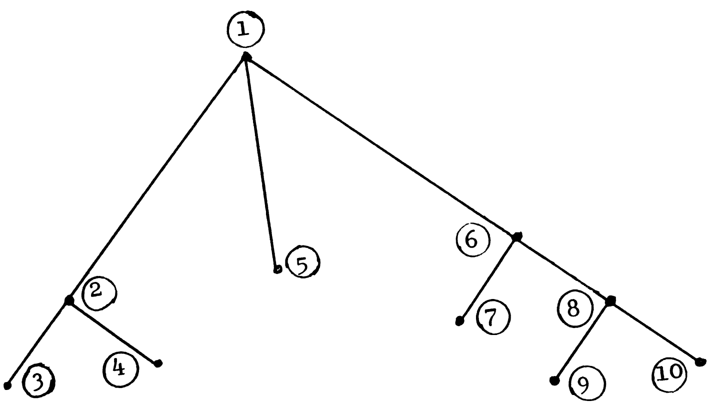


Fig. A1

Node 2 is the *eldest son* (first son) of node 1.
Node 6 is the *youngest son* (last son) of node 1.

Transversing the tree from the head node the *yo-yo path* through the nodes is that indicated by the ascending numbering.

Each node of a tree has a data stack associated with it. This stack is empty when the node is added to the tree. Subsequently any number of the following items in any order can be placed in this data stack.

(*a*) an integer	(held directly)
(*b*) a stack	(held as a pointer)
(*c*) a string	(held as a pointer)
(*d*) a tree	(held as a pointer)

The facility (*d*) is not used by the inputter but is given for completeness. A string in the ADMIN system is not that defined in the ALGOL 60 report. However, ALGOL 60 type strings are often converted into ADMIN type strings but *not* vice-versa.

Book reviews

Job Control Language and File Definition, by Ivan Flores, 1971; 268 pages. (*Prentice-Hall Inc.*, £6·25)

This is a disappointing book. There is certainly a need for a book which explains job control language for the IBM 360 series in a way which reveals the underlying structure of its semantics, rather than concentrating on its rather arbitrary syntax; and this book sets out to do just that. Unfortunately Professor Flores' book falls a long way short of success.

The idea is sound: to introduce the operating system and its components, and file structure; interleaved with chapters which describe the associated JCL, and attempt to link the two together. The execution is disastrous. The one thing above all others which a book of this sort must be is accurate. Next most importantly it must be clear. Finally it should be complete on its chosen level. This book is none of these things.

A first reading revealed no less than one hundred and ten errors of various sorts—mostly printers errors, which in themselves could be fatal to a book on JCL, but also errors of fact. Particularly prone to error are the examples, which frequently contain faults which would cause strings of diagnostics from the assembler or the reader/interpreter, e.g.

'DISP = (OLD,MOD)' and 'OPEN (IND,OUTD)'.

Where clarity is concerned, Professor Flores and I have opposite views. He seems to believe that the clarity of a text improves in direct proportion to the number of different typefaces in use. I do not; particularly when the same word in different typefaces is used to mean different things.

The decision as to how much information should be provided on each topic is a difficult one for any author. Professor Flores says in his preface that his book 'will be a reference source for system programmers' and 'find good application in a senior or graduate course in computer science'. In my opinion he provides far too little information for the former group. As to the latter . . . well, in view of the other shortcomings of this book, it hardly matters.

Martyn Thomas (London)

Computer Control in Process Industries, by E. I. Lowe and A. E. Hidden, 1971; 279 pages. (*Peter Peregrinus Limited*, £4·00)

This is a project-oriented textbook sponsored by the Council of the Chemical Industries Association and produced by two members of its Instrumentation Advisory Committee, to meet a need which was not satisfied by previous books in this area. Encouraged by such a need, the authors have produced a well-written and easily read textbook, which, whilst primarily aimed at satisfying the needs of the chemical engineer and instrumentation manager, will be of considerable interest to computer professionals working in the field of process control. Whilst those sections dealing with computers—in particular Chapter 3—will be elementary to those involved with computers on a day-to-day basis, the remainder of this book is concerned with two main areas of considerable importance—the interface between the computer system and the process, and the organisation and management of computer projects in this particular field. Several portions of Chapter 10 concerned with the organisation of computer projects closely resemble portions of the BCS Code of Good Practice, although written independently.

Summarising, this is a book which deserves a place on the bookshelves of all concerned with the control of continuous processes, in the light of the wide range of training and reference material which it contains; ranging from central processors, through interfaces such as CAMAC and BS 4421, to management and codes of practice.

F. E. Taylor (Manchester)