A ring structure processor for a small computer

By N. E. Wiseman and J. O. Hiles*

A low-level data structure package for the PDP7 computer is described. Its principal features are the compact form in which given structures may be set up and the wide range of formats permitted. Ring structures are regarded as special cases of general list structures, and the package permits the generation and processing of all legal list structures. It is, however, specifically oriented to a certain class of uni-directional list and ring formats, and achieves particularly good space utilization when they are used. Space statistics for the package are presented which, when compared with the performance of more conventional schemes, show typical savings of about 30%.

Introduction

The work described in this note was carried out as part of an on-line circuit design project now in progress utilizing a PDP7 computer and 340 CRT display as an interactive terminal on the Cambridge multiple-access computing system operating on the Titan computer. For reasons to do with operator convenience, it is required to maintain a part of the data structure representing the task in hand in the PDP7 as well as in the Titan. The ring structure processor (RSP) was designed to facilitate this while providing for simple translation rules between the structures maintained in the two machines. The structures running in Titan are represented in ASP (Gray, 1967). Particular attention has been paid to the shortage of core store in the PDP7, and processing time and user convenience (in terms of freedom from programming restrictions and side effects) have been deliberately sacrificed in RSP to obtain economy in storage space. The package is based on a modified version of SLP, a simple list processor for the PDP7 (Wiseman, 1966), and adopts the same storage allocation scheme.

Particular features of RSP

Data structures in RSP are constructed from *data cells* and *non-items* each occupying one 18-bit word. The successor word to a data cell is in the next higher consecutive address while the successor to a non-item is given by the address in the non-item itself. Non-items are regarded as an implementation feature invisible to the user, who simply accesses, via the package, sequences of data cells. The 13-bit address field in each word has a meaning determined by the remaining 5 digits according to **Table 1**. The significance of the terms will become apparent later.

When setting up a structure the package issues words from free storage as required (including non-items when necessary) to provide suitably connected sequences. For example, the 3-atom sequence DOG may turn out in many different ways such as those shown in **Fig. 1**, depending on the available free storage.

In the first case shown in Fig. 1, consecutive words happened to be issued from free storage and each word was thus available for use as a data cell. In the last case, non-items were required to chain together pairs of available words resulting in a LISP-like format of two words per data cell. The typical case also shown in the figure would be somewhere between these two extremes, requiring sometimes one word, sometimes two, per data cell. The successor to a particular data cell is known, as in LISP, as its CDR (pronounced "cudder") and the operations of forming and accessing successor cells are performed by a CDR-function in the package in such a way that the user is unaware of any intervening non-items that may arise.

Table 1

5-BIT IDENTIFIER	13 BIT ADDRESS FIELD
00000	Atom
00010	Atomic name
00100	Address if non-zero Terminator if zero (NIL)
00110	Ringstart
x1xxx	Pending (garbage collection in process)
10000	Non-item

Lists are referenced indirectly by means of registers (known as *base registers*) held outside the list storage area. Symbolic names may be associated with these registers using the PDP7 Assembler in the usual way. Two classes of names are distinguished, temporary and permanent. Lists are normally initiated by declaring their names to the package, which remembers them for later use by the garbage collector. These are the *permanent names*. It is the user's responsibility to leave permanent names at the head of structures he wants to

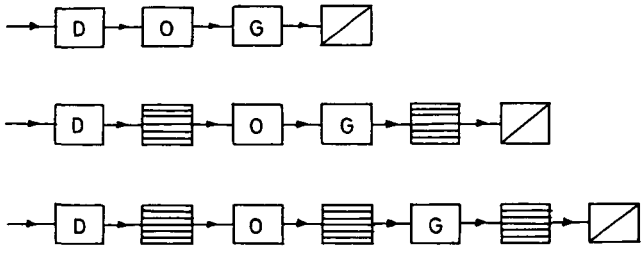

Fig. 1.—3-Atom sequences

* *University Mathematical Laboratory, Corn Exchange Street, Cambridge.*

preserve, as otherwise the garbage collector may return them to the free list without notice. In processing a structure additional names may be attached and moved about the structure to speed access to particular areas. These are the *temporary names* (known sometimes as "bugs") and are not taken account of by the package in determining accessible structures. Except in this respect, however, the package does not discriminate between temporary and permanent names and there is practically no limit to the numbers of each which may be used.

The elementary operations of the SLP list package, on which RSP is founded, enable lists to be initiated, extended, edited and traversed. In RSP there are in addition a number of subroutines for dealing with certain formal structures made up from sets of *elements*. These elements are simply lists in a particular format which are used to manufacture the "building blocks" required in a data structure for the formation of hierarchical ring structures. The first few cells of an element, known as the *head*, are used to hold the structural information. Each cell in the head contains either the start of a ring (*ringstart*) or the address of the next member of the ring (*ringpointer*). The user may, if he desires, ignore the actual connection mechanism used in RSP and merely consider the elements of his data structure as if they were composed of contiguous 13-bit registers. The number of cells and their order in the head is arbitrary. Accordingly a mechanism is provided to define the extent of the head and this function is performed in RSP by a cell of type *atomic name* (abbreviated to *atname*) which terminates the head. The value of this cell is then used as the element name for all communications within the program. The user, however, may associate an arbitrary character string, known as the *printname*, with each element and the printname is then stored in a list referenced by the atname in the manner shown in **Fig. 2.**

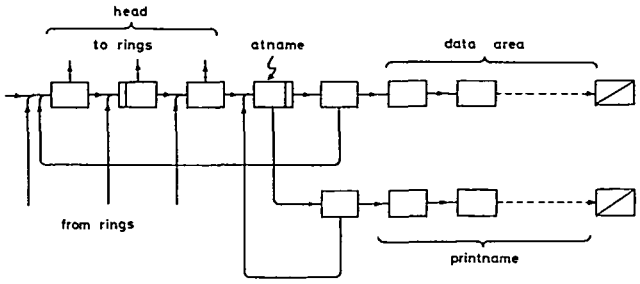

Fig. 2.—Typical element

Owing to the fact that the head is not necessarily composed of adjacent registers in store but is a one way list, a pointer to the top of the element head is provided. This pointer follows the atname in the element as shown in Fig. 2. The list provided for user information (data) about the element follows the head and the whole element

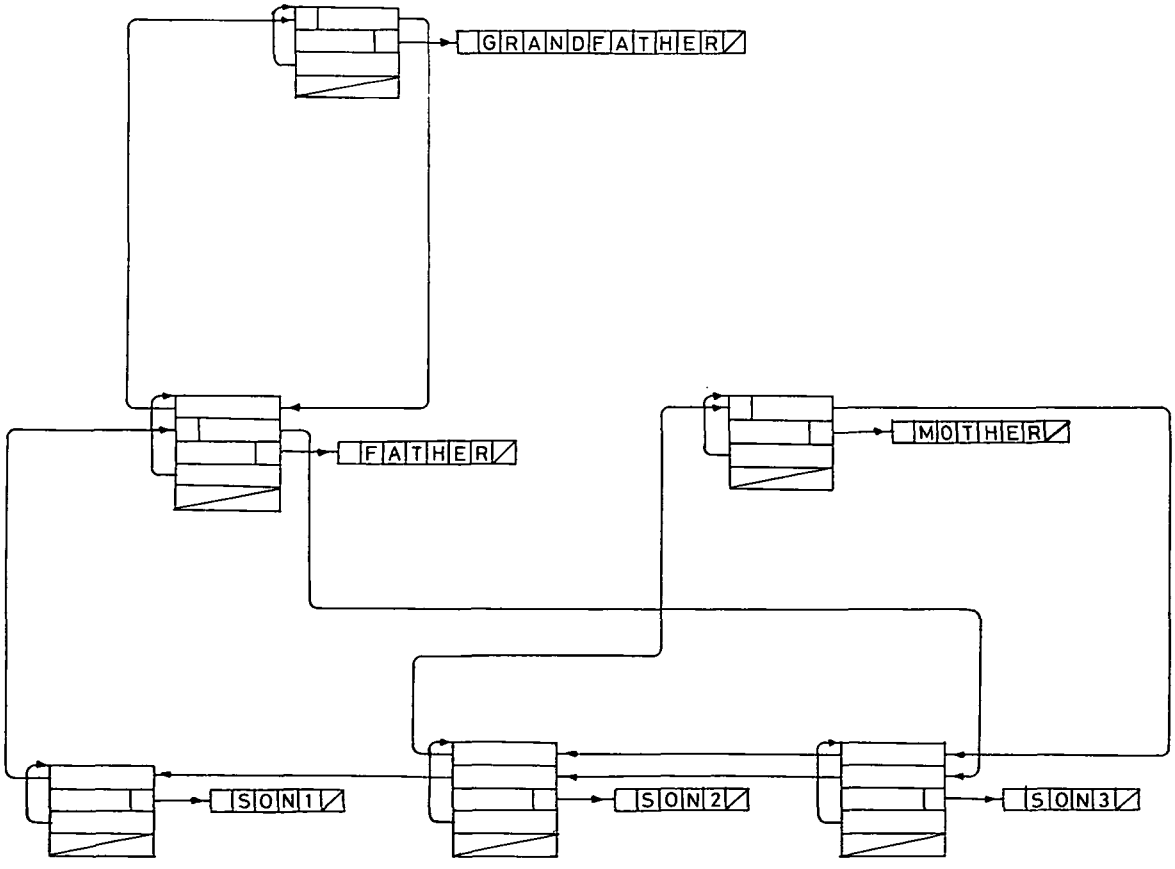

Fig. 3.—Simple family tree

is terminated by a NIL cell in the usual way. In Fig. 2 a vertical bar in the left hand of a data cell indicates a ringstart and a vertical bar in the right hand of a data cell represents the atname. Each ring formed from connected elements has exactly one ringstart and the element in which the ringstart is situated is said to own both the ring and all other elements on it. Two kinds of data may be associated with each ring:

(1) A *name* or *type descriptor* for the ring. This data item is regarded as belonging to the ringstart cell and is put into the data area of the element carrying the ringstart cell.

(2) *An associator* which gives a value to the relationship between a ring member and the ringstart element. Each ringpointer may have an associator which is stored in the data area of the element carrying the ringpointer.

Thus each ringstart or ringpointer in the head of an element may carry a *qualifier*, in the form of a name or associator respectively in the data area. Since there is likely to be more than one pointer in the head it is necessary somehow to distinguish between the different qualifiers. This is done by stipulating that when all qualifiers are present, the package assumes that they occur in the top of the data area in the reverse order to the head cells they qualify. A simple counting operation then allows the qualifier to be obtained for any head cell. In cases where the qualifier is not explicitly required, it may be omitted to save space but then, of course, it is the user's responsibility to administer the data area.

As a simple example of the RSP ring system, consider the small fragment of family tree shown in **Fig. 3.** Interconnecting non-items are omitted for clarity, but otherwise the conventions of Fig. 2 are adopted. It will be seen how seniority of one member of a relational ring is established over the others by means of the ringstart cell. Thus FATHER "owns" his sons through the *brother* ring into which they are tied by ringpointers. Starting at any brother, father can only be found by going to each successive element until the ringstart is encountered, since it is only possible to traverse the ring in one direction.

The data structure handling program is not in general concerned with the user's name for an element and may use the atname itself to identify the element. However, the user will normally gain access to the element via the printname since, as the system is implemented at Cambridge, this is less likely to be relocated within the computer core store. The first entry of the printname list is therefore a pointer to the atname of the element.

Details concerning each member of the family tree can be added in the data area of the element as a list structure of any complexity, providing that the user is prepared to administer it. In simple cases the qualifier facility may be used with the particular list format already described.

At this point it seems convenient to introduce a shorthand notation for the pictorial representation of ring

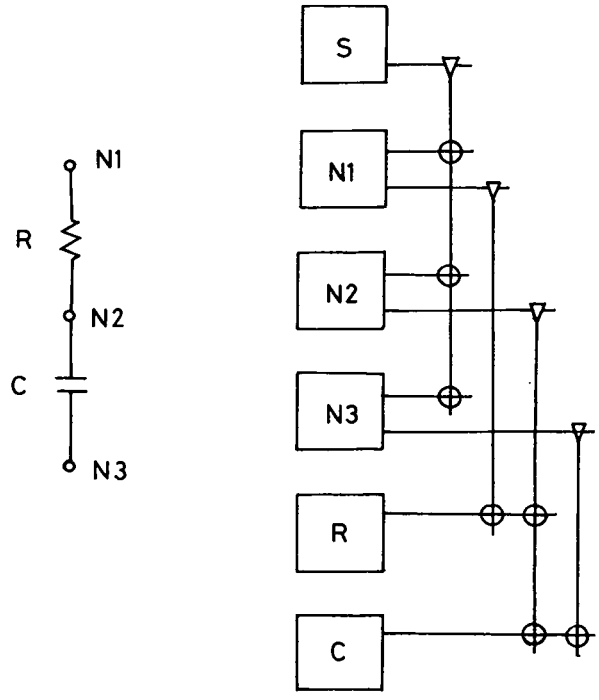

Fig. 4.—Ring structure notation

structures. The notation was originally proposed for the representation of structures built with the ASP system (Gray 1967: 1966), but is attractive for general use in data structure representation. (RSP is a low-level relative of ASP.) Boxes represent elements (actually non-head parts of RSP elements), triangles are ringstarts, and circles ringpointers. Lines represent traversable connections amongst the elements, those joining with boxes being CDR chains (the heads of elements), the remainder being CAR chains (the rings themselves).

A simple structure representing a particular interpretation of a resistor-capacitor network is shown in **Fig. 4.** The S-element is the top level representation for the circuit and may be said to own it via the three nodes N1, N2 and N3. These sons of S in turn own the network branches (components) R and C and are therefore linked into a ring of S as brothers while being fathers of other elements in their own right. To simplify the diagrammatic representation of the structure, rings in the notation are not shown completed but are terminated by a small perpendicular bar. Thus N1, N2, N3 are on the ring owned by S, while R is on two rings, one owned by N1 and one by N2. Information about the branches and nodes (characteristics, position, etc.) could be contained in the data areas of the respective elements. However, this is not pertinent to the structural representation and is not shown in the diagram.

Since elements are formed as CDR chains of data cells they may be edited and extended freely using normal list-processing methods. All allowable ring structures are in fact perfectly legal list structures and may be handled as such by the package. The view is taken that ring structures are simply special cases of list structures

and the package has therefore been designed basically as a list processor with ring handling attachments. In particular the free storage is administered in the same way as in LISP. Available registers are chained together forming the *free list* from which data cells are issued to the user as required. When the free list expires, the structures accessible to the user (via his permanent names) are examined and any space remaining is chained up to form a new free list. This process, known as *garbage collection*, is performed automatically by the package when necessary and the user is normally not aware of its occurrence. In RSP the garbage collection routine is rather more involved than in the conventional case since two sorts of garbage may have to be handled. First there is the usual sort mentioned above arising from words which have become inaccessible to the processing program. Second there is what is known as *litter* existing in the structure in the form of chained redundant non-items (one non-item is always sufficient to connect a data cell with its successor). This litter arises during the processing of a structure and can multiply without limit.

A number of possible side effects in the use of the package should be noted. Names which are left attached to a list which is being processed may get changed, disconnected, or set to "impossible" values (i.e. so as to point to non-items). Such a situation may occur for instance when the subroutine POP is used on a list. The function of this subroutine is to remove an item from a specified point and to "close up" the list. This is done by copying the following cell into the one to be removed and inserting a non-item into the next word pointing to the remainder of the list. It will be appreciated that if the list was originally composed of consecutive cells, a cell will have been converted into a non-item by this process so that any name set to this will now be pointing at a non-item. There is, as in WISP (Wilkes, 1964), no protection against side effects except careful programming. They are, in the main, a direct consequence of adopting a system which permits a multiplicity of names to coexist on one structure.

The interpretation of words in the list area is determined explicitly by the 5 non-address digits in each word. It is not permitted to infer the meaning from context and there is thus no provision for handling arbitrary 18-bit user words within the data structure. When such words are required they must be stored (by the user) outside the list area and referenced indirectly from the structure via atoms. An atom is thus regarded as either a *value* or an *indirect value* whereas an address is always an *indirect data cell.*

Implementation details

The user of the package will normally employ a mixture of machine code orders and package calls in forming and processing structures. He must thus remain aware of the detailed arrangement of his data and of the exact behaviour of the package in different circumstances.

An area in core declared by the user is allocated for the use of the package for list storage. A call to the package (SETUP) is then made to initialize various settings and form the free list, after which structure building can begin. If, while processing a structure from within any package subroutine, the free list expires, then automatic garbage collection takes place in an attempt to form a new free list. It takes place in three phases.

In phase one a pass is made through the allocated list storage area and all litter items are disconnected.

In phase two each declared list is scanned and marked by setting bit 1 in every word visited. A stack is used to save the start and branch points of each list during processing.

In phase three, a final pass through the list storage area is made returning all unmarked words to the free list and removing the marks.

If this is successful, the interrupted subroutine is resumed and eventually control transfers back to the user program in the usual way. The user is aware of only the longer-than-usual time delay. If, however, the branch list expires in phase 2 or if no garbage is collected in phase 3, control transfers instead to a special location where a user recovery routine may be entered to attempt some restart process.

Usually one or more list names are supplied by the user as parameters for each call to the package. These names are handed over in the form of literals (on the PDP7 as LAW PIG for the list named PIG etc.), the base registers which are addressed by the names having been assigned at assembly time in the form of unset variables. In this way any group of up to 6 characters which forms a valid unassigned name to the assembler forms also a valid list name for RSP.

The contents of the base registers are addresses of registers in the list area and are stored in the format of address items (00100 in the 5 non-address bits). Operations such as CAR A := B therefore consist simply of a copy

```
LAC B     /Load accumulator with contents of B
DAC I A   /Deposit accumulator as contents of contents
            of A
```

Lists are terminated by a NIL data cell (100000) which is supplied automatically by the package as required. Names which are moved over the structure by the user provide the normal means for him to inspect and write his lists. A name which arrives on a terminating NIL would ordinarily allow the user to unterminate the list with some operation like DAC I⟨NAME⟩ without the package being able to stop him. In anticipation of such action the package therefore arranges that, when it has moved a name up to a NIL cell it extends the list with an additional NIL terminator before returning control to the user. Lists are thus always properly terminated and the user is free to write in any cell, including NIL, through a name.

The primary objective in implementing RSP was to minimize storage requirements for a given structure

while retaining the processing flexibility of more conventional schemes. With infinite core and a perfect algorithm, structures could be built utilizing slightly over one word per data cell. A conventional list processor on the other hand would require exactly two words per data cell. RSP structures would normally lie somewhere between these two, although in particularly unfavourable cases they could use considerably more than two words per cell (until the garbage collector is called, litter may accrue in the structure). Tests have been made on the package to estimate its actual utilization of space under typical conditions. Sequences of elements of random length were created and disconnected in the presence of a growing list, whose words were scattered through the list area, until the garbage collector failed to retrieve any useful space. The length of each element created was compared with the minimum possible length (one word per data cell). The results are shown in **Fig. 5.** Prior to the first entry to the garbage collector length extensions of less than 25% occur. After successive entries the space utilization worsens, gradually at first, more rapidly later as the chance diminishes of finding consecutive sequences of registers. Some 10 or 12 collections occur before 100% extension (the figure for a conventional scheme) is passed, at which point roughly 90% of available space is occupied by the scattered list. Now the package fails even to find pairs of consecutive registers to form into item/non-item units. However, in practice, it is very unlikely that it would prove necessary to call the garbage collector so often and to have to contend with such a large and scattered list.

Facilities provided by RSP

The package, as written for the PDP7, is intended to be assembled with the user's program. Calls which the user's program makes to the package are in the form of subroutine jumps, which appear in assembly code to the user as ⟨FUNCTION NAME⟩. Calls are usually preceded, and occasionally followed, by the name of a list which is used as a parameter for that call. Some 25 calls exist at the time of writing, providing simple list operations (traverse to CAR, traverse to CDR, pushdown, etc.), operations with elements and rings (form an element, insert element in ring, find ringstart, delete a ring, etc.), and operations which assist recursion in user programs (enter a routine, exit a routine, send operand to stack, etc.). Some choice in the use of the operations is possible. Element deletion, for example, is regarded as an application-sensitive function and provision is therefore made for a variety of deletion algorithms to be set up by the user from basic routines in the package. The need for different deletion procedures may be demonstrated with the aid of the simple structure shown in Fig. 4. Consider the node N2. If branch R or branch C should be deleted, node N2 would remain as the termination point of the other branch, C or R respectively. However, if both R and C were deleted, the node N2 would lose its significance and hence may be deleted as well. On the structure shown this would be termed "upwards" deletion and would correspond with the removal of elements having no offspring (i.e. owning no rings). Conversely, it is evident that the deletion of node N2 would require the deletion of branches R and C (since it leaves them incompletely specified) and this

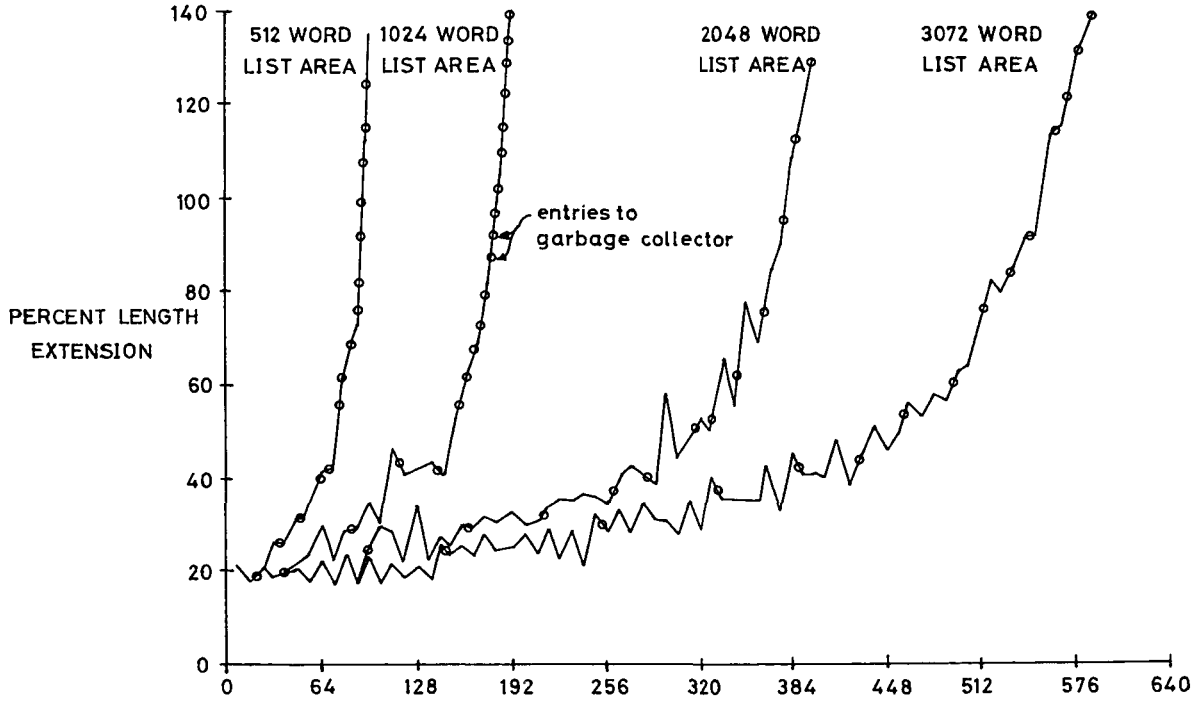

Fig. 5.—RSP space utilization

operation, known as "downwards" deletion, corresponds with the removal of all descendants of a given element. Provided only branches may be the subject of a deletion call, upwards deletion will suffice. However, it is clear that if nodes may be the subject, a "both-ways" deletion is required. It is left to the reader to discover structures which, in given applications, would require downwards only deletion, or simple one-level deletion in which descendants vanish only by "falling off" the accessible structure. Any of these schemes can be implemented by the user from RSP routines as required.

The debugging of a program which uses RSP is assisted if the structures built by the program can be inspected conveniently by the user. An experimental program, known as The Entertainer, has been written to display selected parts of an RSP structure on the screen of the 340 CRT display connected to the PDP7 at Cambridge. Its action is as follows. The user calls the Entertainer and supplies the address of any cell in the head of the element at which he wishes to start probing. A block representing the head of the element is then displayed by the program at an origin situated in the north-west corner of the CRT screen. A triangle represents a ringstart in the head while a circle represents a ringpointer. The atname is associated with the body which is displayed as a rectangle. **Fig. 6** shows a typical structure as displayed by an interim version of the program. A cross is also displayed initially at the top of the block and the lightpen is sensitive to this (and nothing else except a "light button") so that tracking may occur. Thus the lightpen may be moved about on the screen and, providing any part of the cross may be "seen" by the pen, the former is repositioned to be centred on the lightpen. The block in turn moves with the cross so that it may be positioned anywhere on the screen by the user. The user may now switch off the lightpen to prevent tracking and switch it on again when he has pointed it at a special stick protruding from the bottom of the block (the light button). This causes the tracking cross to disappear from the screen and the lightpen to become sensitive to the sticks protruding from the cells of the block. When one of these is "seen" by the lightpen by pointing the latter at it, the stick and its cell are

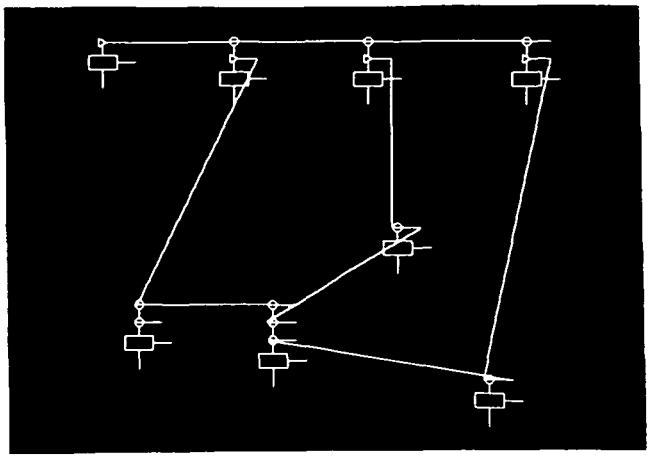

Fig. 6.—The entertainer display

intensified on the screen and the program notes both the cell in the data structure which has been selected and the cell pointed at by it. The user can now examine the contents of these using the PDP7 debugging program, DDT. The user may change his mind as many times as he likes by pointing at other sticks until a final choice is made when the light button is again pointed at by the lightpen. The program will now display the new element at the end of the stick complete with a tracking cross and a line joining it to the previous block, so that it may now be moved about on the screen with the lightpen as before, the interconnecting line being amended as appropriate. When stick activation occurs again, any stick on any block may be selected and further blocks displayed on the screen to build up a pictorial data structure. If the program detects that an element chosen for display by stick selection is already shown as a block on the screen, then instead of creating a new block (a duplicate), an interconnecting line is drawn to the existing one.

Acknowledgements

The authors wish to acknowledge the assistance given by Crispin Gray through many helpful discussions, and by John Grant and Quentin Van Abbé with implementing the package on the PDP7.

References

BOBROW, D. G., and RAPHAEL, B. (1964). A comparison of list-processing computer languages, *Comm. ACM*, Vol. 7, p. 4.

MCCARTHY, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Pt. 1, *Journ. ACM*, Vol. 3, No. 4, p. 84.

ROBERTS, L. G. (1964). Graphical communication and control languages, Lincoln Laboratory MIT Reprint MS1173.

WILKES, M. V. (1964). An experiment with a self-compiling compiler for a simple list-processing language, *Annual Review of Automatic Programming*, Vol. 4, Pergamon Press.

WISEMAN, N. E. (1966). A simple list-processing package for the PDP7, *Proc. 2nd European Seminar, Digital Equipment Corporation Users Society.*

SUTHERLAND, I. E. (1963). Sketchpad, a man-machine graphical communication system, *AFIPS Conference Proceedings, Spring Joint Computer Conference.*

GRAY, J. C. (1967). Compound data structures for C.A.D.—a survey, to appear in *Proc. 20th Anniv. Conference of the ACM, Washington.*

GRAY, J. C. (1966). Revised specification for the Cambridge Data Structure Package, Privately circulated.

NEWMAN, W. M. (1967). A method of control for interactive programs. Privately circulated.

ROSS, D. T. (1961). A generalized technique for symbol manipulation and numerical calculation, *Comm. ACM.*, Vol. 4, No. 3, p. 147.

KNOWLTON, K. C. (1966). A programmer's description of L6, *Comm. ACM*, Vol. 9, No. 8, p. 616.

Appendix 1

Operating the package

The package is supplied in a symbolic form ready to be assembled with the user's program. It makes no references to absolute addresses except for the default settings of parameters which define the workspace and list area. These parameters should normally be set by the user program at run time prior to initializing the package. They are

BEG address of start of list area (default setting 2000)
END address of end of list area (default setting 3000)
BOTN address of bottom of branch stack (default setting 3001)
TOPL address of top of branch stack (default setting 3200)
ERR + 1 jump instruction to recovery routine (default setting HLT).

N.B. ERR is a location in the package to which a subroutine jump occurs if some error is discovered. BOTN and TOPL define working space for the garbage collector and storage space for declared names.

Calls to the package are in the form ⟨FUNCTION NAME⟩ which assembles into a JMS ⟨SUBROUTINE ADDRESS⟩. Most calls require one parameter, usually the name of a list, and this is loaded into the accumulator in the form of a literal LAW ⟨NAME⟩ prior to the call. Exceptionally a subroutine may require two parameters and in this case the second parameter is placed in the address following the call, also in the form LAW⟨NAME⟩. The list of subroutines and calling sequences which follows comprises the list of tested features at the time of writing. Further high-level calls will be added as usage develops.

Call	*Effect*
SETUP	Initialize package. Form free list, empty list of declared names, etc.
LAW ⟨NAME⟩ INIT	Initialize a list and attach to ⟨NAME⟩. Add ⟨NAME⟩ to list of declared names.
LAW ⟨NAME⟩ CDR	⟨NAME⟩ := CDR⟨NAME⟩. If an attempt is made to move past a NIL item the chain is automatically extended. Exits with LAW⟨NAME⟩ in Acc.
LAW ⟨NAME⟩ CAR	A simple subroutine which follows the pointer given by ⟨NAME⟩, chaining past any non-items to the true CAR. There is no test as to whether an atom is being operated on.
LAW ⟨NAME⟩ PUSH	A list is pushed down at the point specified by ⟨NAME⟩, i.e. an element inserted while the connections with the preceding and following sections of list are maintained.
LAW ⟨NAME⟩ POP	This pops up or removes an element from a list at the point specified by ⟨NAME⟩ while preserving the connections between the rest of the list.
LAW ⟨NAME⟩ ENTER	Enter routine called ⟨NAME⟩. Return address is saved on stack called LINK in package.
EXIT	Exit from routine to address on stack.
LAC ⟨OPERAND⟩ STAK	Send operand to stack called LOP in package.
UNSTAK	Load Acc. with operand from stack.
FEL	Form an element with head length one. Exits with EL1 pointing to head, EL1 in Acc., null ring in head word and printname list empty.
LAC (N FELN	Form an element with head length N + 1. Exists with EL1 pointing to head, EL1 in Acc., null rings in all head words and printname list empty.
LAW ⟨NAME⟩ NULLR	Form null ring with ringstart in ⟨NAME⟩. Exits with LAW⟨NAME⟩ in Acc.
LAW ⟨NAMEQ⟩ INSRT LAW ⟨NAMEP⟩	Insert ringpointer at ⟨NAMEQ⟩ in ring at ⟨NAMEP⟩. Return does not skip over second operand so LAW⟨NAMEP⟩ is replaced in Acc.
LAW ⟨NAME⟩ ADDW	Insert a word at head of element. On entry ⟨NAME⟩ is anywhere in header. On exit ⟨NAME⟩ is on new word, new word contains a null ring and Acc. contains LAW⟨NAME⟩.
LAW ⟨NAME⟩ FINDS	Move ⟨NAME⟩ round ring to ringstart. Exits with LAW⟨NAME⟩ in Acc.
LAW ⟨NAME⟩ FINDN	Move ⟨NAME⟩ down element to atname. Exits with LAW⟨NAME⟩ in Acc.
LAW ⟨NAME⟩ FINDQ	Find qualifier for headword at ⟨NAME⟩. Exists with value of qualifier in Acc. and OP + 1 on address of qualifier. If during traverse of data area a NIL cell is encountered, it is replaced by the atom ZERO (000000).
LAW ⟨NAME⟩ DSON	Delete a son ring, ⟨NAME⟩ on ringstart (i.e. father).

LAW ⟨NAME⟩ DELB	Delete element from brother ring at ⟨NAME⟩, i.e. destroy the father/son relationship with the element containing the ringstart.
LAW ⟨NAMEX⟩ CONC LAW ⟨NAMEY⟩	Concatenate two elements, ⟨NAMEX⟩ and ⟨NAMEY⟩ pointing to atnames, resulting element to have atname from ⟨NAMEY⟩. Return does not skip over second operand so LAW⟨NAMEY⟩ is replaced in Acc.
LAW ⟨NAME⟩ RIDS	Destroy ringpointer at ⟨NAME⟩ together with any redundant structure up to, but not including, start element.
LAW ⟨NAME⟩ ELDS	Destroy element on ⟨NAME⟩ together with any redundant structure up to, but not including, start element.
LAW GRHA ENTER LAW ⟨NAME⟩ LAW ⟨FN1⟩ LAW ⟨FN2⟩	Go round the head of the element moving the cell pointer *after* applying function. Execute function ⟨FN1⟩ if a ringstart is encountered or function ⟨FN2⟩ if a ring-pointer. ⟨NAME⟩ is on any item in the head.
LAW GRHB ENTER LAW ⟨NAME⟩ LAW ⟨FN1⟩ LAW ⟨FN2⟩	Go round the head of the element moving the cell pointer *before* applying function. Function ⟨FN1⟩ is executed for a ringstart and ⟨FN2⟩ for a ringpointer. ⟨NAME⟩ is on any item in the head.
LAW GRRA ENTER LAW ⟨NAME⟩ LAW ⟨FN⟩	Go round a ring moving the cell pointer on *after* applying function. ⟨NAME⟩ is on the ring.
LAW GRRB ENTER LAW ⟨NAME⟩ LAW ⟨FN⟩	Go round a ring moving the cell pointer on *before* applying function. ⟨NAME⟩ is on the ring.

Error recovery

In applications where core space is in short supply, or where the parameters of the package have been inappropriately set, frequent error exits from the package may occur. In some cases, recovery is possible and the link address in ERR will indicate what course of action to try. A listing follows (address in octal):

INIT − JMS + 7	means no room on branch stack for holding the name declared in an INIT call. To recover try increasing TOPL and returning with JMP I ERR.
EXIT − JMP + 3	means the link list is empty; i.e. more calls to EXIT than to ENTER have been made. This is a user error.
GARB1	means the list storage area is fully occupied with active lists. To recover increase END, write zeros into the freed space (i.e. from old value to new value of END) and return with JMP I ERR.
GARB2	means no consecutive words were found by the garbage collector in phase 3. The chance of continuing with JMP I ERR is small but non-zero. If it fails then increase END, write zeros into the freed space and return with JMP LIM+1.
MK1 MK2	means no room on branch stack for saving an address generated during garbage collection phase 2. To recover, try increasing TOPL and returning with JMP I ERR.

Appendix 2

Examples of the use of RSP

As a simple example, consider the structure of Fig. 4. Assuming that the elements have been created and that their head words are identified with the list names s, N1, N2, etc., the structure could be set up as follows:

```
LAW C
INSRT
LAW N3      /C on N3 ring
CDR
INSRT
LAW S       /N3 on S ring
LAW C
CDR
INSRT
LAW N2      /C on N2 ring
LAW R
INSRT
LAW N2      /R on N2 ring
CDR
INSRT
LAW S       /N2 on S ring
LAW R
CDR
INSRT
LAW N1      /R on N1 ring
CDR
INSRT
LAW S       /N1 on S ring
```

(In practice, of course, list names would not correspond with printnames but the procedure would be the same.)

As a further example of the use of the package, consider the problem of element deletion. Upwards deletion is provided in the package by the function ELDS. Downwards deletion can be carried out with a routine which finds, and applies DELB to, all ringpointers in all descendent elements. The basis of such a subroutine is shown below in the form of two mutually recursive traversing routines which together find and put on a stack all ringpointers in all descendent elements. The

calling sequence is LAC X, STAK, LAW HTRAV, ENTER with X being anywhere in the head of the parent element.

```
HTRAV,  UNSTAK      /traverse head
        DAC Y       /deposit accumulator as contents
                     of Y
        LAW Y
        FINDN
        CDR
        CAR         /move Y to top of head
        LAC I Y     /load the contents of the contents
                     of Y
        AND (140000 /logic AND octal literal 140000 into
                     Acc.
        SAD (140000 /skip if Acc. different from 140000
        JMP . 10    /jump if ringstart to 10th octal
                     location on
        SAD (40000
        EXIT        /exit if traverse complete
        LAC Y
        STAK        /save ringpointer on stack
        LAW Y
        CDR         /next in head
        JMP . -12   /loop, jump to 12th octal location
                     back
        LAC Y
        STAK
        LAC Y
        STAK        /save ringstart
        LAW RTRAV
        ENTER       /traverse ring
        UNSTAK
        DAC Y
        JMP . -13   /loop
RTRAV,  UNSTAK      /traverse ring
        DAC Z
        LAW Z
        CAR         /next in ring
        LAC I Z
        AND (140000
        SAD (140000
        EXIT        /exit if traverse complete
        LAC Z
        STAK        /save ringpointer
        LAC Z
        STAK
        LAW HTRAV
        ENTER       /traverse head
        UNSTAK
        DAC Z
        JMP . -16   /loop
```

Book Review

Machine Intelligence 1, edited by N. L. Collins and D. Michie, 1967; 278 pages. (Edinburgh: *Oliver and Boyd*, 63s.)

This book reports the proceedings of the first Machine Intelligence Workshop organized by Professor Donald Michie at the University of Edinburgh in September 1965. Here the term "machine intelligence" covers a wider field than the better-known term "artificial intelligence". Heuristic problem solving, analogies between human perception and machine pattern recognition are there, but in addition there is also a survey of mathematical methods for proving theorems about particular programs, and papers on compiler-compilers. There are seventeen contributions, far too many to summarize individually. Work in the field can be roughly classified into three areas, theorem proving heuristics and combinatorics, information classification and retrieval, and extensions to programming and display techniques aimed at improving man-machine communications.

Work in theorem proving, game playing, graph theory, and other combinatorial heuristics is well represented. Progress in the area has never been spectacular, and improved algorithms are the usual result. An exceptional step forward was the introduction of the Resolution Principle by J. A. Robinson, and an excellent exposition of it is given here. There is no paper dealing directly with information classification, storage and retrieval. At first this is surprising, for none of the contributors seems unaware of the ultimate importance of large scale, mechanized, data handling. Then the reader begins to make guesses, but more of this later.

Better man-machine communication can take place at many levels. The simplest level is to provide more effective programming languages; the classic example here is the development of LISP to program the Advice-Taker. Several ideas for advancing compiler techniques are described here. The next level is to make programming more flexible. This can be done by giving the user an on-line console, and visual displays, and then letting the computer answer back. Some interesting, but limited, experiments are described. The ultimate level is reached when the computer can respond to voice, or visual, signals in a way resembling that of a human, and several contributors discuss the difficulties and propose solutions to some of them.

This book gives an excellent, fair, snapshot of work in the field in the middle of the decade *in Britain*. There, at the end, is the vital, and disturbing, qualification. Across the Atlantic there is a ferment of work on man-machine communication, much more extensive than that described here. There is also a large effort in the field of information retrieval, although a sobering re-assessment of its value took place recently. The contrast is certainly not a reflection of a disparate quality of thought, or energy; it is caused quite simply by a lack of machines in Britain. Very few researchers in Britain have the use of large, random access, files; even fewer have interactive, personal, computer consoles. Surely, the personal console will become the focus, and stimulus, of work on man-machine communication for the next few years. Adding a 1967 postscript to a 1965 meeting it is worth noticing that Edinburgh University is indeed experimenting with more than one system for personal consoles; but elsewhere one must still note a severe general shortage of up-to-date machines.

J. J. FLORENTIN (London)