A KDF9 ALGOL list-processing scheme

By J. G. P. Barnes*

This paper describes a scheme whereby list processing may be performed in KDF 9 ALGOL by the inclusion of a set of declarations in a program.

1. Introduction

Certain problems now being solved by digital computers deal with objects which are not of the form normally encountered in the fields of numerical analysis, engineering design, etc. These problems deal with objects whose structure is not static, and thus pose a storage problem.

In ALGOL (Naur, 1963) the only structures allowed are single items and multi-dimensional arrays. The arrays must, moreover, be of a fixed number of dimensions and, in KDF 9 ALGOL in particular (see Duncan, 1962), be of fixed bounds for each activation, i.e. no dynamic own arrays. The ALGOL string is also a static structure, only explicit strings being allowed (the use of a string as a formal parameter is also static since it will at call be replaced by an explicit string).

The storage problem is usually solved by the introduction of the concept of a *list*. A list is an ordered sequence of an arbitrary (dynamic) number of objects, which may themselves be lists. This structure is stored in a computer by a technique known as *chain link* storage.

Several programming systems have been devised for list processing but many of them are difficult to learn and use. In general they constitute a separate language and have a separate compiler. The languages themselves are usually not good for general algebraic purposes and the compilers do not offer good diagnostic facilities. There is a lot to be gained by extending an existing language thereby retaining full algebraic and diagnostic facilities. An example of this is ALP described by Cooper and Whitfield (1962).

The system to be described in this paper has features in common with LISP described by McCarthy (1960). The system is written in KDF9 ALGOL and consists of a set of declarations to be inserted into a program. The system is easy to learn and use, and the full facilities of ALGOL are of course still available.

A good introductory description of list-processing techniques is given by Woodward and Jenkins (1961).

2. Lists and atoms

Lists and atoms are defined in Backus normal form as follows

⟨atom⟩ ::= ⟨integer⟩ | ⟨nil⟩

⟨element⟩ ::= ⟨list⟩ | ⟨atom⟩

⟨open list⟩ ::= ⟨element⟩ | ⟨open list⟩, ⟨element⟩

⟨list⟩ ::= (⟨open list⟩)

The objects of our lists are therefore either other lists or integers. There is in addition the special nil atom which will be denoted by N. Examples of explicit lists are

(6 , 9)

(((*N*)))

(1 , (2 , 3) , 4)

Note that the empty list is not defined, i.e. a list has at least one element. This is not an inconvenience; in practice N is used in those cases where an empty list might otherwise arise.

We will use identifiers to represent lists and atoms.

3. Elementary functions

McCarthy defines the following elementary functions (notation slightly different).

atom (x)	where x is an element; true if x is an atom, false if x is a list.
eq (x, y)	where x, y are atoms; true if they have the same value, false otherwise and is not defined if x and y are not both atoms.
hd (x)	the head of x. This is defined if x is a list and is the first member of x. Note that *hd* (x) may be a list or an atom.
tl (x)	the tail of x. This is defined if x is a list and is the list obtained by removing the first element from x. If x has only one element it is the atom N.
cons (x, y)	where x is an element and y is a list or N; constructs the list whose head is x and whose tail is y.

Examples

x	*hd*(x)	*tl*(x)
(6 , 9)	6	(9)
(((*N*)))	((*N*))	*N*
(1 , (2 , 3) , 4)	1	((2 , 3) , 4)

4. A KDF 9 ALGOL list system

Without rewriting the compiler it is impossible for the identifiers used for lists and atoms to be other than of the types normally available. All list elements are in

* *Imperial Chemical Industries Limited, Bozedown House, Whitchurch Hill, Reading, Berkshire.*

fact denoted by identifiers of type **integer**. The value of this integer indicates where the element is stored.

Storage space for the list elements is provided by two **integer arrays.** The size of the arrays is set by the user in the integer store size. We thus have the declaration

integer array *item, link* [0: *store size*] ;

The size of the store set aside for list elements is therefore fixed in each application. We define the cell i as the pair of variables *item* [i] and *link* [i]. The element x is then stored in cell x as follows. (In the sequel we will refer to "a list x" or "the list whose address is x" meaning the list stored in item [x] and link [x]).

x is a list	*item* [x] is address of head of x
	link [x] is address of tail of x
x is an atom	*item* [x] is the value of the atom
	link [x] $= -1$

Atoms and lists are thus distinguished by the value of the link.

The element N is "stored" at address 0. Both item and link are -1. Thus

$$item\ [0] = link\ [0] = -1$$

The last element of a list thus has link equal to 0. This convention is very useful; it allows Boolean expressions such as

$$tl\ (x) = 0$$

meaning "the tail of x is *atom* N".

Cells not in use are formed into a list known as the *free storage* list. The item of each cell is -1 and its link is the address of its successor. The last cell of the list has its link equal to zero. The address of this free storage list is stored in **integer** *next cell.* Whenever another cell is required **integer procedure** *new cell* is called; this has value equal to *next cell* and sets *next cell* equal to its (old) link. At the start of any use of the system all the cells of the store are placed on the free storage list by a call of the **procedure** *initiate.* Note that cells not in use are distinguished by having item $= -1$. Several checks are applied to ensure that cells on the free storage list are not inadvertently manipulated by the user.

Example

Suppose that store size = 8 and that the store is as follows

i	*item* [i]	*link* [i]
0	−1	−1
1	2	3
2	6	−1
3	4	0
4	9	−1
5	−1	6
6	−1	7
7	−1	8
8	−1	0

Then we say that

cell 0	contains N
1	(6 , 9)
2	6
3	(9)
4	9

and cells 5, 6, 7, 8 are on the free storage list.

The number of cells required to store a general list is

$$2a + n + b - 1$$

where

a = number of atoms excluding N

n = number of N

b = number of pairs of brackets.

Example

(1 , (2 , N) , 3) requires $2 \times 3 + 1 + 2 - 1 = 8$ cells.

This may seem extravagant but it should be noted that duplicated atoms or sub-lists need only be stored once.

(1 , 1) could require 4 but needs only 3 cells.

((7, N) , (7 , N)) could require 8 but needs only 5 cells.

5. Elementary procedures

The facilities of McCarthy's elementary functions are provided by the following ALGOL procedures.

boolean procedure *atom* (x)	is **true** if x is an atom **false** otherwise.
boolean procedure *equal* (x, y)	is **true** if x and y are both atoms or both lists and in the former case they have equal value, and in the latter if they are lists of identical structure whose corresponding atoms are equal in value. Note that N is equal only to itself.
integer procedure *hd* (x)	is address of head of x, i.e. *item* [x]. If x is not a list a fault is indicated. (N.B. If x is a cell on the free storage list the fault is also indicated. This remark applies to all operand checks.)
integer procedure *tl* (x)	is address of tail of x, i.e. *link* [x]. If x is not a list a fault is indicated.

integer procedure *cons* (x, y) is the address of a new list whose head and tail are x and y. If y is not a list or N a fault is indicated. Each time *cons* is called a cell is removed from the free storage list.

Because of the way in which atoms are stored instructions are necessary for finding their value and for constructing them.

integer procedure *value* (x) gives the value of the atom x. If x is not an atom a fault is indicated.

integer procedure *setup* (x) is the address of a new atom whose value is x. Each time *setup* is called a cell is removed from the free storage list.

Note that *value* (x) and *hd* (x) are essentially the same thing.

6. Input–output facilities

integer procedure *read list* (*dv*). This procedure reads an explicit list in from device *dv*, constructs it and sets *read list* equal to its address. The list must be punched in the form

⟨list⟩;

where ⟨list⟩ is as defined in Section 2. All atoms must be explicit integers defined thus

⟨sign⟩ ::= − | ⟨empty⟩

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩

⟨integer⟩ ::= ⟨sign⟩ ⟨unsigned integer⟩

Editing symbols are ignored. If the data object does not conform to the above description a fault is indicated.

procedure *write list* (*dv* , *f* , *x* , *n*) prints the list (or atom) x on device *dv*, the values of atoms being printed in format *f*; *n* is an integer equal to the length of the format.

Each list is started on a new line. If it cannot all be printed on one line then it will be continued on the next line. In the interests of clarity each line is not necessarily completely filled. The last symbol on an intermediate line will be a comma and will be one of the highest-order commas on that line. Each list is terminated by a semicolon. A page width of 70 characters is assumed.

The nil atom is represented by N for the purposes of these procedures.

7. Other procedures

The following procedures are also provided for the manipulation of lists.

procedure *set hd* (x, y) replaces the head of the list x by element y. If x is not a list a fault is indicated.

procedure *set tl* (x, y) replaces the tail of the list x by element y. If x is not a list or y is not a list or N a fault is indicated.

The above two procedures could be used to construct a cyclic list, e.g. *set tl* (x, x). Such lists are not to be encouraged. They cannot be represented explicitly in a finite form and hence cannot be output by **procedure** *write list*.

procedure *set value* (x, y) changes the value of atom x to y. If x is not an atom a fault is indicated.

procedure *join* (x, y) joins list y on to the end of list x, i.e. it replaces the tail of the last item of x (which is N) by y. List x is thus altered.

integer procedure *copy* (x) is the address of a new list which is a complete copy of the list x apart from its atoms. The list x is not destroyed.

Note that *equal* (*copy* (x), x) is always **true** but *copy* $(x) = x$ is **false** unless x is an atom.

integer procedure *rev* (x) is the address of a new list whose elements are in the reverse order to x. Sublists of x are, however, not reversed.

So if x is (1 , (2 , 3) , 4) then *rev* (x) is (4 , (2 , 3) , 1)

integer procedure *append* (x, y) is the address of a new list similar to that obtained by *join* (x, y). List x in this case is not altered.

8. Storage retrieval

Each time that procedures *cons* or *setup* are used a new cell is taken from the free storage list. It is conceivable that the free storage list would soon be exhausted unless steps were taken to return unwanted cells to it.

Every cell that is accessible to the program is so because it can be reached by a sequence of *hd* and *tl* operations on one of the integers used for storing list addresses. When the contents of one of these integers is changed it may happen that the cell whose address it formerly contained can no longer be reached by a sequence of such *hd* and *tl* operations. Such a cell is inaccessible and is automatically returned to the free storage list as follows.

From among those integers which the user is going to use for the storage of list addresses he selects a set which will be called *base lists*. They are designated base lists by the declaration of **procedure** *bases*. The body of *bases* consists of statements of the form

preserve (x)

where x is a base list. As an example, suppose that x, y

and z are designated base lists, then bases will be declared as follows:

procedure *bases*;

begin *preserve* (x); *preserve* (y); *preserve* (z) **end**;

When the free storage list is exhausted storage retrieval is carried out by **procedure** *tidy*, a call of which performs the following operations. Firstly, *bases* is called which calls in procedure *preserve* with argument equal to the base lists in turn. *preserve* changes the link of each cell that can be reached from its argument in such a way that the cell is distinguished. In fact it sets

$$\text{link} := -\text{link} - 3.$$

Normal cells thus have link $\geqslant -1$, marked cells have link < -1. If a cell is encountered whose link < -1 then it assumes that that cell has already been reached. After procedure *bases* has been called all cells whose links are still $\geqslant -1$ are deemed unwanted and are returned to the free storage list. Finally the links of all wanted cells are restored to their original values. If the free storage list is then found to be still empty a fault is indicated.

The above description is not complete, for consider the following situation. Suppose that the statement

$$x := F(cons(a, 0), cons(b, 0));$$

is being executed where F is some integer procedure with two arguments called by value. The two arguments are then evaluated in order. Suppose that after *cons* (a, 0) has been evaluated the free storage list is exhausted. Storage retrieval will be initiated as soon as *cons* is called during the evaluation of *cons* (b, 0). The cell containing *cons* (a, 0) which has been left hanging will then unfortunately be returned to the free storage list.

This is overcome as follows. A list with identifier *expression* is formed. Each time a new cell is obtained from the store it is added to the list *expression*. This in fact means that two cells are necessary. One is the new cell explicitly required by the user, the other is used to extend *expression* and points to (i.e. its item is address of) the required new cell. The statement

preserve (*expression*)

is inserted in procedure *tidy* immediately after the statement which calls in *bases*. So when storage retrieval occurs all items formed during the evaluation of expressions are not lost. As soon as the ALGOL expression is completely evaluated the list *expression* is set to zero. To determine the end of the ALGOL expression a counter (stored in **integer** *cons count* and set to zero by *initiate*) is increased by one each time a relevant procedure is entered, and decreased by one on exit. The increase is arranged to occur before the arguments of the procedure are evaluated by calling them by name, and transferring to local variables after the increase. If after the counter is decreased it is found to be zero then the ALGOL expression is complete, the list *expression* is set to zero and the cells of it containing the addresses of the new cells it was guarding are returned to the free storage list.

The increase and decrease of the counter are controlled by procedures *ante* and *post*. The general form of a procedure is thus

(i) call procedure *ante*
(ii) transfer arguments to local variables
(iii) body of procedure
(iv) call procedure *post*

It is not in fact essential to do this in the case of procedures which do not explicitly use new cells and have only one argument; *e.g. hd, tl.*

The user is urged to use *ante* and *post* in his own procedures if necessary. The important points to note are

(i) *ante* and *post* must occur in pairs, in that order
(ii) they may be nested
(iii) their use does not prevent all storage retrieval but only the retrieval of those cells constructed since the counter was last zero.

After each storage retrieval **procedure** *warning* is entered. This must be provided by the user and may be used to print a message if desired, e.g. the number of cells now on the free storage list could be printed; this number is stored in *free cells.*

The user should note that if store size is only just sufficient for his problem then a great deal of time may be spent in storage retrieval.

9. Diagnostic facilities

Various checks are incorporated in the procedures, and if one of them fails **procedure** *achtung* will be entered. The procedure prints a message on device *fail dev*. If the **boolean** *fail pm* is **true** it will then give a post mortem of the store apart from cells on the free storage list. This post mortem takes the form of three columns of integers, each row being *i*, *item* [*i*], *link* [*i*]. Finally the statement *sqrt*(−1) is encountered. This terminates the run and may be used to give such diagnostics as the operating system allows. For example, using a Whetstone compiler (see Duncan, 1962) a retroactive trace will be produced. The first few items of the trace refer of course to *achtung* but the remainder should be useful in locating the fault. Note that many procedures call other procedures, e.g. *hd* calls *atom*, so the trace should be interpreted with care

Example: $x := cons(hd(y), tl(z))$

will produce the following (forward) trace

cons
ante
new cell
hd

atom
tl
atom
post

10. Incorporation of system

The system consists of a set of declarations and must therefore be inserted in a block head. In addition the **integer** *store size* must be declared and have a value assigned to it in an outer block.

Assignment must be made to **boolean** *fail pm* and **integer** *fail dev* and the **procedure** *initiate* called once before executing any statements using the system.

The user is advised to declare procedures *bases* and *warning* immediately after the system. Note that all integers preserved by *bases* must also be declared in the same block head.

11. Summary of identifiers

Identifiers used by the system are listed in **Table 1.** They must not be used for other purposes.

12. Text of system

The ALGOL text of the system is not reproduced in this paper, but a copy may be obtained from the author by anyone interested. This text, which naturally provides an exact description of the system, should be referred to for further details regarding points not completely explained above.

13. Example of use of system

To illustrate the use of the system a program to demonstrate the tree sort method of sorting will be described. This method is described by Sussenguth (1963) and an ALGOL procedure implementing it is described by Kaupe (1962). The implementation now to be described is intended to be illustrative rather than practical.

The numbers to be sorted are stored one per node at the nodes of a tree as follows. Each node has two branches leading from it—the left and right branch (either or both may be missing). Each branch leads to a sub-tree, the left or right sub-tree. Each node has the property that all numbers stored in the left sub-tree are less than the number at the node, which in turn is less than (or equal to) all numbers stored in the right sub-tree. This structure may then be added to as follows. The number to be added is compared with the number at the root, if the new number is smaller take the left branch, else the right, and repeat until a node is reached with an appropriate missing branch. The new number is then placed at a new node and the branch added to connect it to the tree.

Having added all the numbers to the tree they are unpacked by first unpacking the left sub-tree then the node and finally the right sub-tree. Sub-trees are unpacked by further applications of the same rule.

Table 1

Reserved identifiers

IDENTIFIER	TYPE	DESCRIBED IN SECTION
store size	**integer**	4
next cell	,,	4
free cells	,,	8
cons count	,,	8
expression	,,	8
fail dev	,,	9
fail pm	**boolean**	9
item	**integer array**	4
link	,, ,,	4
write list	**procedure**	6
initiate	,,	4
achtung	,,	9
tidy	,,	8
ante	,,	8
post	,,	8
preserve	,,	8
set tl	,,	7
set hd	,,	7
set value	,,	7
join	,,	7
copy	**integer procedure**	7
rev	,, ,,	7
new cell	,, ,,	4
hd	,, ,,	5
tl	,, ,,	5
cons	,, ,,	5
value	,, ,,	5
setup	,, ,,	5
append	,, ,,	7
read list	,, ,,	6
equal	**boolean procedure**	5
atom	,, ,,	5

In our list scheme the trees and sub-trees are represented by lists as follows. The first element is a list representing the left sub-tree (or *N* if absent), the second element is an atom whose value equals the number at the node, and the rest of the list is the right sub-tree.

Numbers are added by the recursive procedure *join on* and the tree is unpacked by the recursive procedure *print*.

The program which follows is self explanatory. The instruction *write list* shows the situation when the tree is complete. Warning messages are produced each time storage retrieval is initiated, and finally, when a sort is attempted on more numbers than the store can handle, the fail routine is entered and a post mortem produced.

The text of the program, data and results are shown in the Appendix.

Acknowledgement

The author wishes to express his thanks to Imperial Chemical Industries Limited for permission to publish this paper.

References

COOPER, D. C., and WHITFIELD, H. (1962). "ALP: An Autocode list-processing language," *The Computer Journal*, Vol. 5, p. 28.
DUNCAN, F. C. (1962). "Implementation of ALGOL 60 for the English Electric KDF9," *The Computer Journal*, Vol. 5, p. 130.
KAUPE, A. F. (1962). Algorithm No. 143. *Comm. Assoc. Comp. Mach.*, Vol. 5, p. 604.
MCCARTHY, J. (1960). "Recursive Functions of Symbolic Expressions and their computation by Machine, Part 1," *Comm. Assoc. Comp. Mach.*, Vol. 3, p. 184.
NAUR, P. (Editor) (1963). "Revised report on the algorithmic language ALGOL 60," *The Computer Journal*, Vol. 5, p. 349.
SUSSENGUTH, E. H. (1963). "Use of Tree Structures for Processing Files," *Comm. Assoc. Comp. Mach.*, Vol. 6, p. 272.
WOODWARD, P. M., and JENKINS, D. P. (1961). "Atoms and Lists," *The Computer Journal*, Vol. 4, p. 47.

Appendix

Example of use of system

PROGRAM

```
  Treesort
begin integer storesize; open (20); open (10);
    copytext (20, 10, [ ; ]); storesize := read (20);
    begin comment

        Insert text of system here;

        procedure warning;
        begin write text (10, [[c] only]);
            write (10, f, freecells);
            write text (10, [**cells*available[c]])
        end;
        procedure bases;
        preserve (tree);

        procedure joinon (i, tree);
        value i, tree;
        integer i, tree;
        if i <value (hd (tl (tree))) then
        begin if hd (tree) =0 then
            sethd (tree, cons (0, cons (setup (i), 0)))
            else joinon (i, hd (tree))
        end else
        begin if tl (tl (tree)) =0 then
            settl (tl (tree), cons (0, cons (setup(i), 0)))
            else joinon (i, tl (tl (tree)))
        end;

        procedure print (tree);
        value tree;
        integer tree;
        if tree≠0 then
        begin print (hd (tree));
            write (10, f, value (hd (tl (tree))));
            print (tl (tl (tree)))
        end;

        integer n, f, tree;

        faildev := 10; failpm := true;
          f := format ([ndd]);
        initiate;

        n := read (20);
    again: tree := cons (0, cons (setup (read (20)), 0));
        for n := n−1 while n>0 do joinon (read (20),
          tree);
        write text (10, [[cc]]);
        write list (10, f, tree, 3);
        write text (10, [[cc]]);
        print (tree);
        n:= read (20);
        if n≠0 then goto again;
        close (10); close (20)
    end
end
→
```

DATA

```
Trial*data;
75;

 5;
 3; 47; 43; 73; 86;

20;
36; 96; 47; 36; 61; 46; 98; 63; 71; 62; 33;
26; 16; 80; 45; 60; 11; 14; 10; 95;
```

```
30;
97; 74; 24; 67; 62; 42; 81; 14; 57; 20; 42; 53;
32; 37; 32; 27;  7; 36;  7; 51; 24; 51; 79; 89;
73; 16; 76; 62; 27; 66;

 0;

→
```

RESULTS

```
Treesort
Trial data
(N, 3, (N, 43), 47, N, 73, N, 86);
 3 43 47 73 86
only 16 cells available

((((((N, 10), 11, N, 14), 16), 26), 33), 36,
((N, 36, (N, 45), 46), 47, (N, 60), 61, (N, 62), 63, N,
   71, N, 80, N, 95), 96, N, 98);

10 11 14 16 26 33 36 36 45 46 47 60 61 62 63 71 80 95 96 98
only 61 cells available

List fail  No more cells available
List post mortem
        1      67     −1
        2      54     70
        3       1      0
        4      24     −1
        5       2     74
        6       4      8
        7      74     −1
        8       5      3
        9       7     72
       10      97     −1

                etc.
```

Book Reviews

The Impact of Computers on Accounting, by T. W. McRae, 1964; 304 pages. (London and New York: *John Wiley and Sons Ltd.*, 42s.)

The avowed object of this book is to interest accountants in the things that can be done with computers.

In early chapters Mr. McRae seeks to outline the basic ideas involved in computers themselves and in their applications. He then passes on to consider Operational Research, the Audit question, and the economics of E.D.P. The rest of the book, the more satisfactory part, is concerned with the impact of E.D.P. on management, particularly the accountants' share in management, the problem of education, and the future demands on the accountant.

The book seeks to cover a wide, perhaps too wide, field, and could have a better title. Indeed at the end of a chapter the author uses the words—"This chapter is concerned with the impact of computers on accounting." The overall target is more probably management, particularly as exercised or influenced by the accountant.

The case for an increased appreciation of the computer is well put, and the extent of current possibilities, on the whole, are well described. On the other hand, despite the obvious efforts to avoid it, the descriptions and in some cases opinions are oriented towards one manufacturer's ideas.

One is surprised to find that punched cards are the sole means of feeding a computer, and paper tape is condemned as slow and relegated to applications where it can arise as a by-product. The needs for random access are stressed and the author doubts the efficiency of exception reporting as a means of minimizing this need.

There is a good attempt at a classified Bibliography, but the Glossary and Index are weak.

Notwithstanding these criticisms the accountant with the patience to follow the arguments may well feel that the author makes his case for the profession to think anew regarding its own basis of training, its system of qualification, and indeed its basic philosophy. He may be less inclined to accept the solutions proposed.

E. C. Lay

Data Acquisition and Processing in Biology and Medicine—Volume 3, edited by K. Enslein, 1964; 344 pages. (Oxford: *Pergamon Press Ltd.*, 100s.)

This volume reports the proceedings of the third Rochester Conference. The subjects covered include diagnostic routines, multivariate analysis as used in diagnosis, literature retrieval problems, machine analysis of heart sounds, limitations of various data-acquisition and analysis methods, and a rather thorough treatment of statistical computer methods for diagnosis. The emphasis is mostly on clinical medicine.

There are several excellent papers on the diagnosis of disease by using what is essentially classificatory statistical techniques and information-retrieval methods. There is more than one claim that computer diagnosis can be made more reliable than human diagnosis, especially in fields where specialists do not encounter very many cases. As one contributor puts it: "In actual usage, the computer has proved a wise colleague to the pediatric cardiologist, and a superior consultant to members of a general hospital staff specifically interested in congenital heart disease."

The book reflects the state of mathematical infiltration into medicine. As yet statistics has made the greatest contribution to medicine, and applied mathematicians have still to make any great contribution via model making and analysis. This is coming in, but this volume, like its predecessors, does not find much space for it.

This is a worthwhile volume, and I can recommend it. The papers on information retrieval and maintenance of case histories can be profitably read by anyone studying computer documentation in general.

Andrew Young