LITHP—an ALGOL list processor

By R. W. L. Trundle*

List Processing has become the primary mechanism for the manipulation of the data structures found in most branches of non-numerical analysis. This paper describes a simple implementation of list processing which can be used on any machine having a suitable ALGOL compiler.

A list processing package was written in collaboration with Mr. P. Hammersley of Northampton College of Advanced Technology for use on the Elliott 503, and the Ferranti Pegasus, though further development was halted on the Pegasus due to lack of store and also on the 503 due to limitations of the Elliott ALGOL. The list processing methods used here depend very heavily on those of Woodward and Jenkins of The Royal Radar Establishment (Woodward and Jenkins, 1961; Jenkins, 1964)—in fact, the processes used in this paper are a slight variation on their exposition of McCarthy's LISP (McCarthy, 1960). The object of this paper is to facilitate the implementation of a list processing procedure on any machine that has an adequate ALGOL compiler, and for descriptive purposes the program is called LITHP. As list processing is essentially recursive, an ALGOL implementation that is not fully recursive itself, such as that of the 503, may impose severe restrictions on the capability of the machine to handle LITHP. However, no such difficulties should occur with machines which have a full ALGOL implementation.

This package consists of sixteen procedures which can be subdivided into four groups:

1. THE CODE PRIMITIVES *type*, *assemble*, *tag*, *extract*, *inword* and *put*, which are used to construct the other procedures of the package.
2. THE LIST PROCESSING PROCEDURES *hd*, *tl*, *cons*, *atom*, *equ* and *null* from which all list processing operations can be constructed.
3. THE HOUSEKEEPING PROCEDURES *clear* and *start*, used for garbage collection and initiation.
4. INPUT/OUTPUT by *inatom* and *out*.

The whole package is nested within two block headings to enable *global variables* and *arrays* to be declared for the package and the user program. Thus the overall structure is:

```
begin    ⟨global variables⟩
  begin    ⟨global arrays⟩
           ⟨LITHP package⟩
             begin   ⟨user program⟩
             end
  end
end;
```

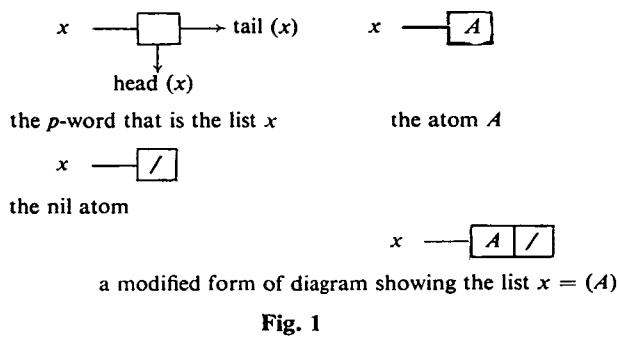


Fig. 1

Machine representation of lists

Two sorts of elements comprise the main list processing store: these are atoms (the basic data to be manipulated) and *p*-words (the link elements that form the structure of the lists). All other words in the store are on the lists of available space. *p*-words and atoms can be shown schematically as in **Fig. 1** where x is the name of a list and A is an atom.

Every word in the stack is divided into three parts:

(i) tag—which consists of a single bit, and in the current implementations this is taken as the sign bit; its use will be discussed under garbage collection.

(ii) type—which consists of two or three bits sufficient to enumerate the types of atoms and *p*-words that are in the stack.

(iii) body—which comprises the remainder of the word and is subdivided according to the integer represented by type:
for a *p*-word, body is divided into two parts (head and tail)
for an atom, body is divided into a number of characters that constitute the atom.

These subdivisions are illustrated in **Fig. 2** with reference to the 39-bit word of the Elliott 503. For a *p*-word, type = 1; for an atom, type = 2; while for a word on the list of available space, type = 0.

A useful extension would be to increase the length of an atom by having two types of atom words, an intermediate and a terminal, so that an atom could become a simple non-branching list.

* *S.H.A.P.E. Technical Centre, P.O. Box* 174, *The Hague, Netherlands.*

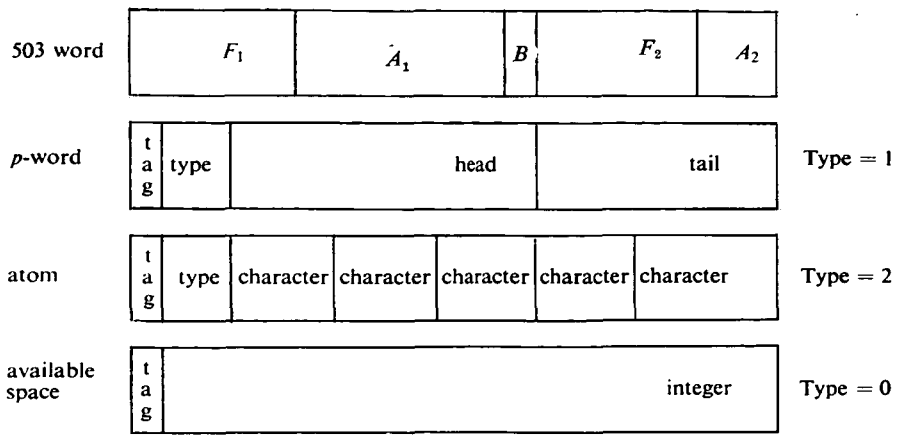


Fig. 2

The global variables and arrays

There are in fact only two global arrays which comprise a vector of names of lists, "*x*", and a main store called "stack" which contains all the information about the lists.

There are four global variables:

(i) *tor*—the size of the vector *x*
(ii) *top*—the size of the vector stack
(iii) *tel*—the stack pointer
(iv) *nil*—the nil atom

In any implementation *tor*, *top*, *tel* and *nil* are, of course, all integer variables.

The code primitives

1. **integer procedure** *type* (*p*); **value** *p*; **integer** *p*;
   comment
   This procedure collates and shifts those bits of the word in the stack that are type, and expresses the result as an integer;
   begin *code* **end**;
2. **integer procedure** *assemble* (*a*, *b*, *c*); **value** *a*, *b*, *c*; **integer** *a*, *b*, *c*;
   comment
   Assemble packs the three integers a, b and c into a single p-word, as the head, tail and type respectively;
   begin *code* **end**;
3. **integer procedure** *tag* (*p*); **integer** *p*;
   comment
   This procedure negates the single bit that is the tag of an element of the stack;
   begin *code* **end**;
4. **integer procedure** *extract* (*a*, *b*); **value** *a*, *b*; **integer** *a*, *b*;
   comment
   Extract gives the integer that is the head or tail of the p-word b according to whether a = 1 or not. If b is not a p-word an error condition occurs;
   begin *code* **end**;
5. **integer procedure** *in word*;
   comment
   This procedure scans the input medium for an atom initial delimiter, and for the 503 *this was taken to be a tild,* ~. *The ensuing characters are then packed into a single word until the word capacity has been reached or a terminal delimiter has been encountered, which for the* 503 *is a space*;
   begin *code* **end**;
6. **procedure** *put* (*s*); **value** *s*; **integer** *s*;
   comment
   This procedure unpacks the characters that comprise the atom s and transmits them to the output medium;
   begin *code* **end**;

List structure

Let *x*, *y* and *z* be list names and *A*, *B*, *C* and *D* be atoms. Use the following notation:

(i) $x = (A, B)$ to mean the list *x* consists of the atoms *A* and *B*.
(ii) $y = C$ to mean that *y* is the atom *C*.
(iii) $z = (D)$ to mean that the list *z* consists of the atom *D*.

Using the modified form of the schematics of Fig. 1, these structures are shown in **Fig. 3.**

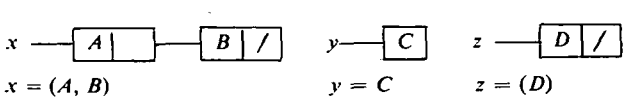


Fig. 3

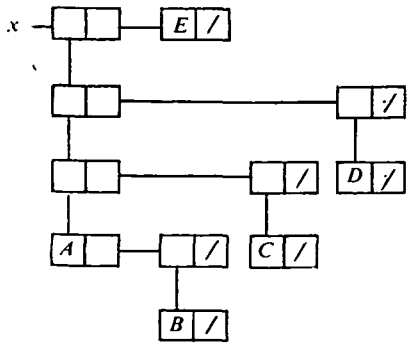


Fig. 4

As the elements of a list can themselves be lists, complex structures can be built from atoms and *p*-words. An example of this is the list $x = ((((A, (B)), (C)), (D)), E)$ illustrated in **Fig. 4.**

The list processing procedures

These six procedures are based on the Woodward–Jenkins definition. They fall into two groups of three: Boolean procedures used to test the status of words in the stack, and integer procedures that are used to manipulate list structures. This latter group can best be defined by means of examples which are illustrated using the schematics of Fig. 1.

(1) if $t = (A, B, C, D)$
then $hd(t) = A$, and $tl(t) = (B, C, D)$

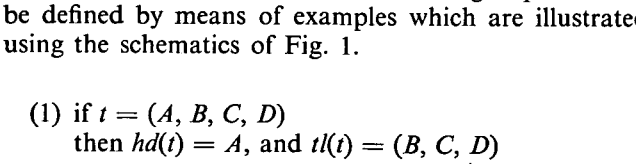


Fig. 5.1

(2) if $u = A$ is an atom
then $hd(u)$ and $tl(u)$ are undefined

(3) if $v = (A)$
then $hd(v) = A$ and $tl(v) = nil$

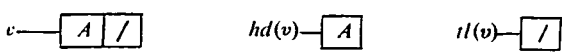


Fig. 5.2

The ALGOL procedures for *hd* and *tl* are:

integer procedure *hd*(*p*); **value** *p*; **integer** *p*;
hd:= *extract* (*1*, *p*);

integer procedure *tl*(*p*); **value** *p*; **integer** *p*;
tl:= *extract* (*2*, *p*):

The ALGOL instructions x:= $hd(x)$; and y:= $tl(y)$; cause the tail and head respectively of x and y to be lost.

(4) if $u = A$, and $w = nil$
then *cons* $(u, w) = (A)$

Fig. 5.3

(5) $u = A, y = (C, D)$
cons $(u, y) = (A, C, D)$

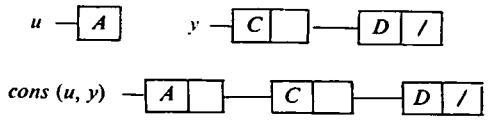


Fig. 5.4

(6) $x = (A, B), y = (C, D)$
cons $(x, y) = ((A, B), C, D)$

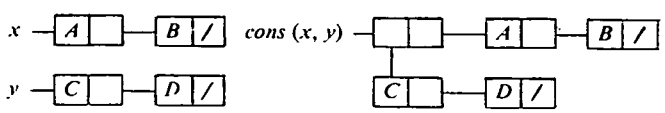


Fig. 5.5

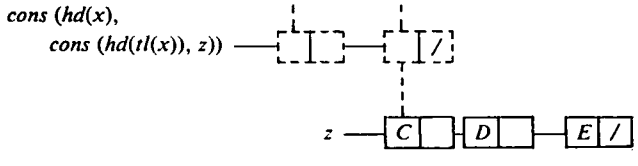


Fig. 5.6

(7) $x = (A, B), z = (C, D, E)$
cons $(hd(x), cons\ (hd(tl(x)), z)) = (A, B, C, D, E)$

The ALGOL procedure for *cons* is:

integer procedure *cons* (*p*, *q*); **value** *p*, *q*; **integer** *p*, *q*;
begin integer *u*; **if** *atom* (*q*) ∧ ¬ *null* (*q*) **then**
begin comment *this is an error condition. The solution adopted was to output an error message and equate the list to the nil atom*;
cons:= 0; **go to** *fin*
end else *cons*:= *u*:= *tel*; *tel*:= *stack* [*u*]; *stack* [*u*]:= *assemble* (*p*, *q*, *1*); **comment** *this takes the next word off the stack, forms it into a p-word and increments the stack pointer*; **if** *tel* = *nil* **then** *clear*; **comment** *if stack exhausted call garbage collector*; *fin*: **end**;

The three Boolean procedures together with their ALGOL implementations are:

(i) *atom* (*A*) *which has a value* **true** *if A is an atom, otherwise* **false.**
boolean procedure *atom* (*p*); **value** *p*; **integer** *p*;
atom:= *type* (*p*) = 2;

(ii) *equ (A, B) defined for any parameter A and an atom B has the value* **true** *if A is the atom B, otherwise* **false.**

```
boolean procedure equ (p, q); value p, q; integer p, q;
    equ:= if atom (q) then stack [p] = stack [q]
    else false;
```

(iii) *null (a) which is* **true** *if A is the nil atom, otherwise* **false.**

```
boolean procedure null (p); value p; integer p;
    null:= stack [p] = nil;
```

Garbage collection

This parameterless procedure retrieves from the stack all words that are not elements of lists and concatenates them into a list of available space. The process is to tag all words on the stack that can be traced from list names; all untagged words can now be formed into a simple non-branching list. The following program gives an explanation of the procedure for doing this:

```
procedure clear;
  begin integer j, k; boolean first; integer array z [1 : 50];
    comment
      An arbitrary assumption has been made that not
      more than 50 branches will need to be held at the
      same time in the auxiliary stack z. This can, of
      course, be varied at will;
    for j:= 1 step 1 until tor do
    begin integer p; p:= x[j]; k:= 1;
    A: if stack [p] ⩽ 0 then
    begin tag (stack [p]);
              if type (p) = 1 then
              begin z [k]:= tl (p); k:= k + 1;
                    p:= hd(p); go to A
              end
    end;
    if k > 1 then
    begin k:= k − 1; p:= z [k]; go to A end
    end;
    for j:= 1 step 1 until top do tag (stack [j]);
    first:= true;
    for j:= 1 step 1 until top do
    begin if stack [j] < 0 then
      begin if first then
        begin tel:= k:= j; first:= false end
        else k:= stack [k]:= j;
        tag (stack [j])
      end
    end;
    stack [k]:= nil
    end;
```

Initiation

This is a single parameterless procedure that must be the first instruction of the user program. Its function is:

(i) to set *nil* as an atom with a zero body;

(ii) to set each list on the vector of lists, *x*, to be a list of a nil atom;

(iii) to concatenate the vector, *stack*, making it a single list of a nil atom.

The procedure is:

```
procedure START;
  begin integer run;
    code; comment to set nil to the null atom;
    for run:= 1 step 1 until tor do
    x [run]:= 0; stack [0]:= nil;
    comment every list is now a null atom;
    for run:= 1 step 1 until top do stack [run]:= 0;
    clear; comment the call of clear concatenates the
    stack and sets the stack pointer, tel, to the head
    of the list that the stack now forms.
end;
```

Input/output

This is of necessity highly machine-dependent but by the use of two code primitives, *inword* and *put*, and the assumption of the procedures *print* ⟨"string"⟩ and *print* ⟨control character⟩, input/output can be described in terms of ALGOL procedures:

Input

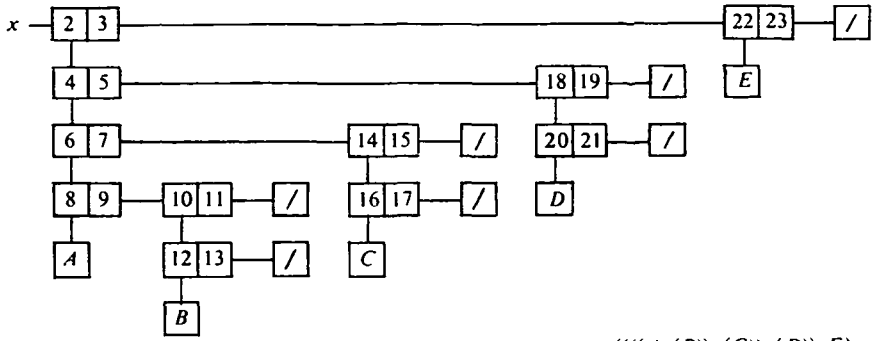
```
integer procedure inatom;
  begin integer mark;
    inatom:= mark:= tel;
    tel:= stack [tel];
    stack [mark]:= inword + nil;
  comment this puts the word on the stack and incre-
          ments the stack pointer;
  if tel = nil then clear;
  comment if the stack is exhausted the garbage
          collector is called.
end;
```

It is only necessary to input the string that comprises a single atom, as all lists are constructed from atoms by use of the list processing procedures. Lists can be built up by use of such operations as *x*:= *cons* (*inatom*, *x*);. An atom can only be attached to the head of a list, so that a list is formed in the reverse order to that in which the atoms are input.

Output

Output is achieved by tracing a list from its head; each time a *p*-word is encountered the address of the tail is put on an auxiliary push-down stack, and the trace is continued from the head of that *p*-word. When no further progress can be made the process goes to the address at the top of the auxiliary stack, and when this is also exhausted the complete list has been output. To obtain a normal list output of the form ((*A*, *B*), *C*, (*D*), *E*) the following conventions are used:

for a *p*-word encountered during a head trace, output (



$x = ((((A, (B)), (C)), (D)), E)$

Fig. 6

for a p-word on the auxiliary stack, output ,
for an atom, output the atom
for a nil atom on the auxiliary stack, output)

To illustrate this consider the list $x = ((((A, (B)), (C)), (D)), E)$, given in Fig. 4, that is shown again diagrammatically in **Fig. 6,** where this list is taken to be occupying locations 1 to 23 of the stack. The integers in the p-words are the head and tail addresses.

Fig. 7 shows the order in which the elements of the list are handled, and the contents of the auxiliary push-down stack at each stage.

LOCATION	CONTENTS	OUTPUT	CONTENTS OF PUSH-DOWN AUXILIARY STACK
1	2 : 3	(	3
2	4 : 5	(	5, 3
4	6 : 7	(	7, 5, 3
6	8 : 9	(	9, 7, 5, 3
8	*A*	*A*	9, 7, 5, 3
9	10 : 11	,	11, 7, 5, 3
10	12 : 13	(	13, 11, 7, 5, 3
12	*B*	*B*	13, 11, 7, 5, 3
13	*nil*	)	11, 7, 5, 3
11	*nil*	)	7, 5, 3
7	14 : 15	,	15, 5, 3
14	16 : 17	(	17, 15, 5, 3
16	*C*	*C*	17, 15, 5, 3
17	*nil*	)	15, 5, 3
15	*nil*	)	5, 3
5	18 : 19	,	19, 3
18	20 : 21	(	21, 19, 3
20	*D*	*D*	21, 19, 3
21	*nil*	)	19, 3
19	*nil*	)	3
3	22 : 23	,	23
22	*E*	*E*	23
23	*nil*	)	——
	EXIT		

Fig. 7

The flow chart, **Fig. 8,** together with the following ALGOL program, gives the general method. The limitations and restrictions of the host compiler will, of course, determine the modifications that may have to be made before implementation.

The procedure is:

```
procedure out (j); value j; integer j;
  begin integer p, c, k, qq, r; integer array z [1 : 50];
    k:=1, c:=0; p:=j; go to A;
    B:z[k]:=tl(p); c:=c+1; k:=k+1; p:=hd(p);
    A: if atom (p) then
    begin qq:= stack [p]; put (qq); c:=c+5;
      C: if k ⩽ 1 then go to fin;
      k:=k−1; p:=z[k]; if atom (p) then
      begin if null (p) then
        begin print (")"); c:=c+1; go to C end;
        print (error message); comment salvage
        routine required.
      end;
      print (","); if c ⩾ 90 then
        begin print (new line); c:=0 end; go to B
  end;
  print ("("); go to B;
fin: end;
```

This procedure was designed for the 503 (but, of course, will not run without modification) with a fixed atom character length of 5 and a Flexowriter line length of 90. The character count is the location c. The assumption of a fixed atom length can be overcome by the introduction of a side effect in the primitive *put*, that records the number of characters in an atom as they are transmitted to the output medium.

Discussion

The hierarchy of the procedures is:

type, assemble, tag, inword, put
extract, atom, null
hd, tl, equ
clear, cut
cons, inatom, start

and will be needed if a requirement of the host compiler is that procedures cannot be called until they are declared, as on the 503.

This system is still under development, and its next implementation may be on a CDC 3600. Any comments or suggestions will be welcome. Possible lines of development are to introduce data tables as well as unlimited length atoms, and to have aliases for the list names since the notation $x\ [i]$ is somewhat clumsy.

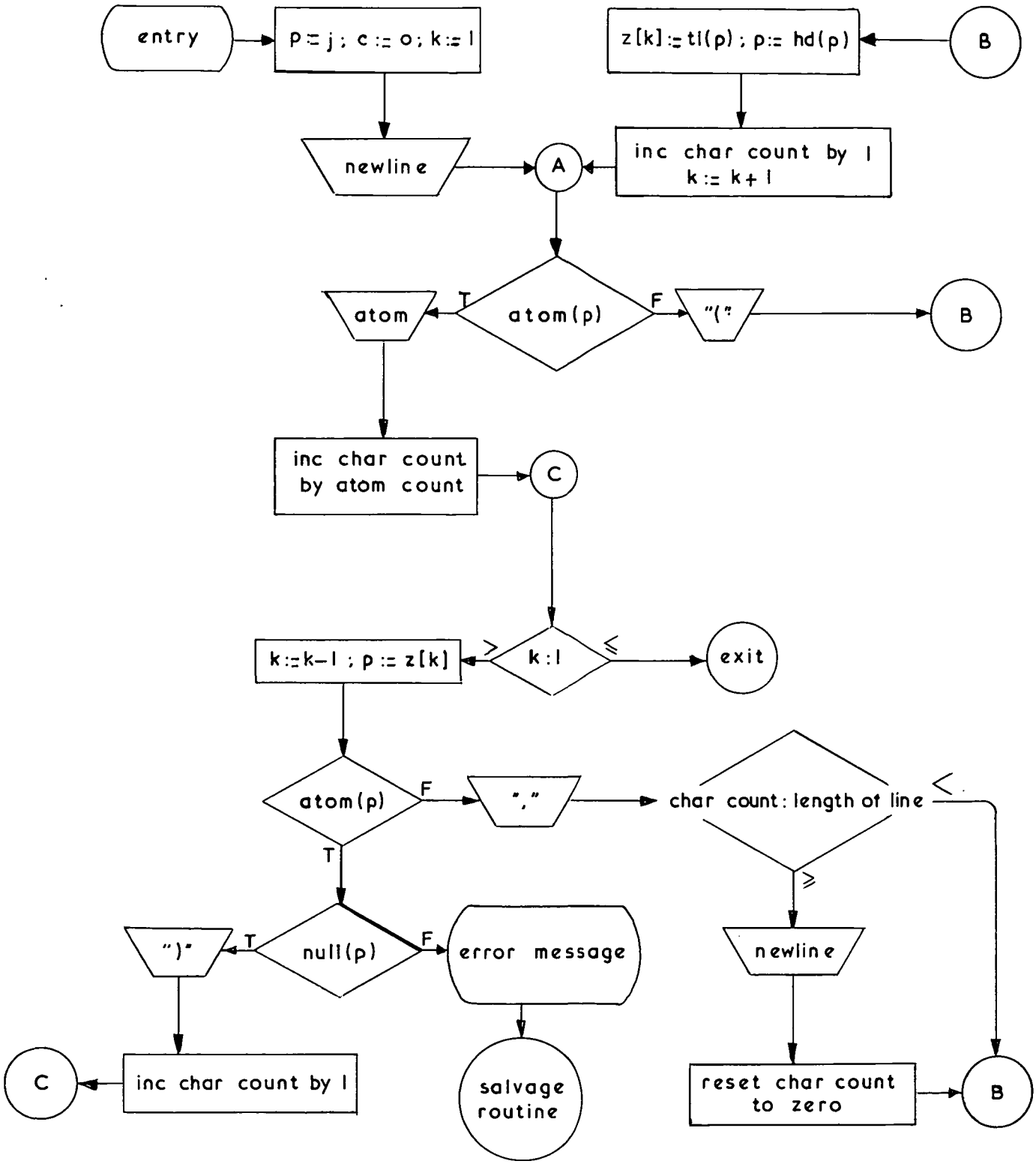


Fig. 8

References

WOODWARD, P. M., and JENKINS, D. P. (1961). "Atoms and Lists," *The Computer Journal*, Vol. 4, p. 47.

JENKINS, D. P. "List Programming," *Introduction to System Programming*, AP 1964, Editor: P. Wegner, p. 238.

MCCARTHY, J. (1960). "Recursive Functions of Symbolic Expressions and their Computation by Machine," Part I, *Communications of the Association of Computing Machinery*, Vol. 3, p. 184.