ALP: An Autocode List-Processing Language

By D. C. Cooper and H. Whitfield

This paper describes an addition to the Mercury Autocode language which enables it to deal with problems requiring list-processing techniques. The new autocode instructions are described and an example is given of a recursive routine.

1. Introduction

The purpose of this paper is to describe an addition to the Mercury Autocode Language so that it may deal with problems requiring list-processing techniques as used in such languages as IPL (Newell and Tonge, 1960) and LISP (McCarthy, 1960; Woodward and Jenkins, 1961). The number of extra Autocode instructions has been kept to a minimum (11) so that this extension is very easily learnt and forms an easy introduction to this technique. The scheme to be described has been implemented on Mercury as an addition to the Autocode, but at present it operates only on cells in the fast store of Mercury and there is thus a restriction to a total of about 500 cells. This of course precludes the use of the scheme in most problems in which it might be useful, but in later machines, such as Atlas or Orion, this scheme could form a powerful addition to the Autocode.

The scheme is designed in the form of eleven instructions added to the existing Autocode Language (Brooker, 1958) so that Autocode instructions may be intermingled with these new symbol-manipulating instructions. Only a desirable minimum of new instructions is included, but subroutines to perform more complicated list manœuvres, such as copying tree structures or joining lists, may easily be built up from these. As one of the authors (D. C. C.) has already produced an IPL interpreter for Mercury it is probable that the language is biased in this direction rather than in the direction of other symbol-manipulation languages.

Even with the storage limitations of Mercury, it has been possible to write a program to perform analytic differentiation of algebraic expressions written in a generalized form of the Mercury Autocode Language. The ALP system has also proved useful in the writing of a statistical survey program. This involved the programming of a simple compiler for which the user defines in a simple verbal language the information on punched cards and the questions he wishes to ask.

It is assumed in the following paragraphs that the reader is familiar with the broad outline of the Mercury Autocode Language.

2. Cells, Symbols and Links

The storage available in the computer is divided up into cells which consist of two parts, a *symbol* and a *link* (a cell will usually occupy one word in the computer).

Symbols are the fundamental units which are manipulated by the language. A number of symbols may be joined together to form a *list*, and the symbols on these lists may be manipulated by means of the new instructions provided. Thus we can find whether a given symbol is on a list; and then we could delete this symbol from the list or perhaps add another symbol either before it or after it. The interpretation of the symbol is left entirely to the programmer (there is no internal marking as to whether a symbol is local or regional, as there is in IPL).

A symbol in ALP consists of a number of *fields* each of which is capable of holding the address of any cell. However, any other desired meaning may be attached to the symbol. In any particular implementation of the language there will be a maximum number of fields allowed. It is not necessary to use all the fields and it is possible to refer to any one or more of the fields separately. (The present version of ALP, written to operate in the fast store of Mercury, allows up to three fields of ten bits each.)

The link of a cell is the address of the next cell on the list. If the cell referred to is the last cell on a list its link is made 0 (zero) to signify this. A link of 1 is used for a special purpose (see Section 7).

3. Representation of Symbols

In an ALP program, whenever we wish to refer to a symbol we write its fields separated by the character /. If the first field only is being referred to we can write this alone, and similarly with two fields if they are the first two, etc.

Thus the following are all allowable forms for referring to symbols in an ALP program:

```
1/20/-4
3
A/I/B23
1/*/J
I
*/*/CJ
I/R
```

The meaning of the character in any field is as follows:

(A) *An Integer* (in Mercury this must be in the range −512 to 511 or 0 to 1,023 according to the programmer's interpretation). The value of this integer is taken as the relevant field of the symbol.

(B) *A main or special variable.* The address of the variable is taken as the relevant field of the symbol.

(C) *An index.* The contents of the index specified are taken as the relevant field of the symbol.
(D) *An Asterisk.* This field is to be ignored in the instruction.

4. The Instructions

Most of the new instructions refer, either directly by name or indirectly by referring to an index containing the name, to a cell which is usually regarded as the head cell of a list; they also refer either to a symbol specified in any of the forms of Section 3, or to a set of one, two, or more indices specified as in Section 3 where only forms (C) and (D) would be allowed. The former would be used in those instructions which write a symbol into a cell, and the latter in those instructions which read a symbol out of a cell. We see, therefore, that all manipulation of symbols is done via the indices.

In the definitions of the instructions which follow we use the following conventions:

cell — stands for the name of the cell operated on. It is specified by writing in the instruction.
(i) A main or special variable.
This is the actual cell to be operated on.
(ii) An index.
The content of the index is taken as the address of the cell to be operated on.
Examples A B44 C(I + 3) J T

symbol — stands for the symbol to be written into some specified cell and may be specified in any of the forms of Section 3.
Examples A/B/20 I/3/J I/*/K A I J/L/M

indices — stands for the one, two or more indices into which the relevant fields of the symbol of the specified cell are to be written.
Examples I/J/K */J T/Q */*/R

index — stands for any index into which the name of a cell to be found will be written.

label — stands for any label number of the current chapter.

The actual instructions are:

TO LIST (cell, symbol)

The symbol in the specified cell is replaced by the given symbol. If any field of the given symbol is undefined the corresponding field of the cell is unaltered.

FROM LIST (cell, indices)

The relevant fields of the symbol of the specified cell are transferred to the indices named.

PUSHDOWN (cell, symbol)

The given symbol is placed in the specified cell, the symbol originally in that cell being transferred to a new cell which is inserted on the list after the specified cell. Effectively all cells on the list whose head cell is the specified cell are moved one down the list, and the given symbol added at the top of the list. The required new cell is taken from a separate list formed of all available spare cells and referred to as the Available Space List.

Table 1

The Effect of the List-Processing Instructions, TO LIST, FROM LIST, PUSHDOWN, POP UP, INSERT AFTER

LIST BEFORE	INSTRUCTION	LIST AFTER	
S_1 S_2 S_3	TO LIST (cell, symbol)	S S_2 S_3	Only alters those fields of the cell which are defined in S
S_1 S_2 S_3	FROM LIST (cell, indices)	S_1 S_2 S_3	S_1 is transferred to the indices named
S_1 S_2 S_3	PUSHDOWN (cell, symbol)	S S_1 S_2 S_3	Undefined fields of S will be left containing junk
S_1 S_2 S_3	POP UP (cell, indices)	S_2 S_3	S_1 is transferred to the indices named
S_1 S_2 S_3	INSERT AFTER (cell, symbol)	S_1 S S_2 S_3	Undefined fields of S will be left containing junk

POP UP (cell, indices)

The relevant fields of the symbol in the specified cell are transferred to the indices named; the symbol in the cell after the specified cell is then transferred to the specified cell, and its link altered to jump over this next cell, which is then returned to the Available Space List. Effectively the symbol in the specified cell is read out, and then all the remaining cells on the list whose head cell is the specified cell are moved up one.

For the effect if the specified cell is already the last cell of a list, see Section 7.

INSERT AFTER (cell, symbol)

A new cell is obtained from the Available Space List, its symbol is set equal to the given symbol, and it is then inserted on the list after the given cell.

The effect of these five instructions is summarized in Table 1 where S refers to the symbol named in the instruction and where, before the instruction is obeyed, S_1 is the symbol in the specified cell and S_2 S_3 . . . the symbols in succeeding cells.

FIND (cell, index, label, symbol)

The list beginning with the specified cell is searched to see if it contains the specified symbol (any fields absent in the symbol are ignored in the test). If such a cell is

found the name of this cell (or of the first such cell if more than one exists) is left in the specified index, and control passes to the next instruction. If no such cell exists the name of the last cell on the list is left in the index, and control is transferred to the specified label.

LINK (cell, index, label)

If the specified cell is not the last cell of a list the name of the next cell is placed in the index, and control passes to the next instruction. If it is the last cell of a list the name of that cell is placed in the index, and control is transferred to the specified label.

SETLINK (cell, cell)

The link of the first cell is set equal to the name of the second cell, i.e. if the first cell is the last cell of a list and the second cell the first cell of another list, then these two lists are joined to form one list.

NEWCELL (Index)

A new cell is removed from the Available Space List, and its name left in the index.

ERASE (cell)

All the cells of the list headed by the specified cell are returned to the Available Space List. Care should be taken if the head cell is referred to by its actual name, e.g. W, since the head cell itself is also returned and may not then be used again in the program.

ADDSPACE (α, β)

α and β may each be any Autocode main or special variable. This instruction is used to add cells to the Available Space List, which starts initially with no cells. All the cells, counting forwards from α to β, are added to the Available Space List. Thus if we wish to use the main variables $A0$ to $A479$ in a list-processing program, we would have the instruction ADDSPACE ($A0$, $A479$) together with the directive $A\rightarrow 479$. (By making use of the fact that, in Mercury, the primed special variables immediately follow the main variables, we could also include those in our Available Space List by means of the instruction ADDSPACE ($A0$, Z').)

When a list-processing instruction calls for a new cell and the Available Space List is exhausted, we get fault 40 (e.g. in Mercury a jump to label 100 of the current chapter, if that exists, or a loop stop with, in either case, SAC containing 40).

The address of the first cell on the Available Space List is kept in a special register (in Mercury in half-register 46).

This is the directive which must be placed at the head of all chapters containing any of these list-processing instructions. It causes all the necessary closed subroutines to be inserted at the end of the chapter.

In Mercury all the above instructions, except ADDSPACE, are compiled into a cue of from 2 to 9 machine orders. For the ADDSPACE instruction a cue of 6 machine orders, to bring the required subroutine to page 15, is inserted in the program. The closed subroutines inserted into the chapter by the ALP directive consist of 154 machine orders.

5. Simple Examples

Two simple examples may make the actual form of these instructions clearer.

(A) Delete from the list named in I all cells which have the integer 1 as the second field of their symbol: leave a count in J of how many such cells there were.

```
      J = 0
22)   FIND (I, I, 21, */1)
      POP UP (I)
      J = J + 1
      JUMP 22
21)
```

(B) Interchange the symbol in cell W with the symbol in the last cell of list W.

```
      LINK (W, I, 10)
 8)   LINK (I, I, 9)
      JUMP 8
 9)   FROM LIST (I, R/S/T)
      FROM LIST (W, O/P/Q)
      TO LIST (I, O/P/Q)
      TO LIST (W, R/S/T)
10)
```

6. Example of a Recursive Routine

Copy the tree structure whose head cell is named in I: leave the name of the head cell of the copy in J.

By a tree structure we mean a list whose members are either pure symbols (to be copied as they stand) or are the names of sublists which are to be copied and the name of the copy to be inserted on the copied list. Each sublist may in its turn contain the names of sublists, and so on, but it is important that the name of no sublist occurs more than once.

This is a simple example of a *recursive routine*, i.e. we shall write a routine to copy a list and, if on this list we encounter the name of a sublist, we shall call in the main routine as a subroutine of itself. In order that this process should work, we must take care of two things. First the "return address" for the routine must be preserved before we call in the main routine; and since the main routine being used as a subroutine may again call in itself, this preservation of the return address must be in a list of return addresses which we may call a *pushdown list*. Secondly, any contents of indices, such as the address of the current cell being copied, which have to be preserved over the routine when it is called in as a subroutine, must also be preserved in a pushdown list. Both these objects are easily achieved with the PUSHDOWN instruction.

We use Z as the head cell of the list of return addresses, i.e. the standard entry to the routine, assuming its first instruction is labelled 1 is

```
T) = return label number
PUSHDOWN (Z, T)
JUMP 1
```

and the standard exit from the routine will be

```
POP UP (Z, T)
JUMP (T)
```

There is one more point to be settled. In the first paragraph above we mentioned the two different kinds of symbols on the lists. There is no such distinction built into ALP (as there is in such languages as IPL): all interpretations of the symbol are left to the programmer. For the purpose of this example we therefore adopt the convention that if the third field of a symbol is -1, then this symbol is the name of a sublist, the address of the head cell of the sublist being in the second field of the symbol.

The actual program is as follows, where we assume that it was entered by the standard entry sequence above.

```
1) NEWCELL (J)          Obtain a new cell for head of
                        copy.
   S = J                S is used to move down the copy.
6) FROM LIST
     (I, P/Q/R)
   JUMP 7, R ≠ −1
   PUSHDOWN             Symbol is name of sublist. Save
     (Y, I/J/R)         the indices I J R and S in 2 cells
   PUSHDOWN (Y, S)      of pushdown list Y.
   I = Q                Set I = name of sublist.
   T) = 2               }
   PUSHDOWN (Z, T)      } Copy sublist.
   JUMP 1               }
2) POP UP (Y, S)        Recover name of copy of last
                        cell.
   INSERT AFTER         In copy place name of copy of
     (S, */J/−1)        sublist.
   POP UP (Y, I/J/R)    Recover I, J and R.
   JUMP 3
7) INSERT AFTER         Not name of sublist.
     (S, P/Q/R)
3) LINK (S, S, 4)       Move S on to last cell copied.
4) LINK (I, I, 5)       Move one down original list.
   JUMP 6               If list not finished go to test and
                        copy next cell;
5) POP UP (Z, T)        otherwise exit.
   JUMP (T)
```

Two further points about this routine should be noted. First, although it makes use of cells Y and Z, these cells are not normally destroyed by the routine. Secondly, it might be thought that an extra blank cell would be at the head of every copied list, since we first generate a blank cell by NEWCELL (J) and the first symbol of the list is inserted after it. However, this is not so; a special marker is put into any cell generated by the NEWCELL instruction, and the INSERT AFTER instruction will then act on this special marked cell as if the instruction were TO LIST. This point will be made clearer in the next section.

7. Private Termination Cells

The ALP language as described in the previous sections is a simple language to learn and to use. However, there is one more complicated feature which must be brought in to deal with certain situations which, in fact, have already been mentioned briefly (see definition of POP UP and also the end of this section).

Consider examples (A) and (B) of Section 6. In these examples the POP UP instruction was used to delete the symbol in a known cell of a list. This is satisfactory unless the cell happens to be the last cell of a list. In this case what is to happen? What should happen is that the previous cell is marked with a zero in its link as the last cell on the list, and then the deleted cell can be returned to the Available Space List. However, this is impossible, as the POP UP instruction does not know the name of the cell linking into the specified cell.

The solution adopted is similar to that of IPL. The cell is not returned to the Available Space List; instead a special marker of 1 is placed in its link. This marker, in effect, signals the fact that this cell does not really exist, and that the previous cell is really the last cell of the list. If, at some future instruction, this private termination cell (to be denoted by ptc) can be returned to the Available Space List and the previous cell marked as a terminating cell, this is done. For example, this operation may arise when a LINK instruction is applied to the true last cell of the list, so that the name of the ptc should occur as output. This is not done; instead the ptc is returned to the Available Space List, the link of the previous cell is set to zero, and the LINK instruction signals that we have reached the last cell on the list.

It should be emphasized that all this is automatic and, in fact, in normal circumstances, there is no need to worry at all about this "non-existent cell"; the list-processing routines will deal correctly with it. However, it may be important to know exactly how each of the instructions deals with a ptc, and so a list of these properties follows.

TO LIST (cell, symbol)

If the cell is a ptc the action taken is exactly as normal but, in addition, the link of 1 is changed to 0 so that the cell is no longer a ptc but is marked as the last cell of the list.

FROM LIST (cell, indices)

It will normally be an error if the cell is a ptc. However, if it is, junk will be left in the indices specified. (On Mercury the B test register is also set > 0.)

PUSHDOWN (cell, symbol)

If the specified cell is a ptc then the symbol is simply placed in the cell, and the ptc marker of 1 in the link changed to the terminating mark zero; i.e. the instruction acts exactly as if it were a TO LIST instruction in this case.

POP UP (cell, indices)

If the specified cell is the last cell of a list (i.e. link is zero) then, after transferring its symbol to the specified indices, it is changed to a ptc by putting 1 as its link. If the specified cell is already a ptc (normally this would indicate an error) then junk will be left in the indices. (On Mercury the B test register is also set > 0.)

INSERT AFTER (cell, symbol)

If the specified cell is a ptc then the symbol is simply placed in the cell, and the ptc marker of 1 in the link is

changed to a zero; i.e. the instruction acts exactly as if it were a TO LIST instruction in this case.

LINK (cell, index, label)

If the cell after the specified cell is a ptc then it is returned to the Available Space List, the link of the specified cell is set to zero, and the LINK instruction then behaves in the normal manner, assuming the specified cell is the last cell of a list. If the specified cell is a ptc (normally this would indicate some error) then the name of the cell is left in the index, and the jump occurs. (On Mercury the B test register is set > 0.)

NEWCELL (index)

The newcell produced is marked as a ptc by setting 1 in its link.

Acknowledgement

The authors wish to acknowledge the useful discussions on this ALP language that they have had with other members of the staff of the Computer Unit, in particular with Dr. M. J. M. Bernal.

References

BROOKER, R. A. (1958). "The Autocode Programs Developed for the Manchester University Computer," *The Computer Journal*, Vol. 1, p. 15.

BROOKER, R. A. (1958). "Further Autocode Facilities for the Manchester (Mercury) Computer," *The Computer Journal*, Vol. 1, p. 124.

MCCARTHY, J. (1960). "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1," *Communications of the Assoc. for Comp. Mach.*, Vol. 3, p. 184.

NEWELL, A., and TONGE, F. M. (1960). "An Introduction to Information Processing Language V," *Communications of the Assoc. for Comp. Mach.*, Vol. 3, No. 4, p. 205.

NEWELL, A., TONGE, F. M., FEIGENBAUM, E. A., MEALY, G. H., SABER, N., Green, B. F., jr., and WOLFE, A. K. (1960). "Information Processing Language V Manual, Sections I and II," Rand Corporation Papers, P1897 and P1918.

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

Book Review

Solutions Numériques des Équations Algébriques, by E. Durand. (Paris: *Masson et Cie*, 1960, pp. 328, 65 N.F.)

Although the title of this book would lead one to believe that it is concerned exclusively with algebraic equations, it is in fact devoted to the solution of single equations in one variable and both algebraic and transcendental equations are treated. As far as algebraic equations are concerned, it is generally assumed that the relevant polynomials are given explicitly; the algebraic eigenproblem, in which the polynomial is expressed in determinantal form, is to be covered in a second volume.

Chapter 1 deals with expansions in power series, the inversion of power series, expansions in continued fractions and similar topics. The methods described here are not likely to be very widely used in practice. Chapter 2 gives a general survey of iterative methods with special reference to first, second and third order processes. The problem of multiple roots is treated in some detail. Chapter 3 is devoted to transcendental equations and discusses the use of iterative methods, particularly that of Newton, and inverse interpolation using the Bessel, Everett and Stirling formulae.

Chapters 4, 5, and 6 are of a more theoretical nature. Chapter 4 discusses the division of polynomials by linear and quadratic factors, the calculation of derivatives, the calculation of the greatest common division and the determination of a polynomial of degree n passing through n points. Chapter 5 is concerned mainly with transformations of polynomial equations and also includes a discussion of the variation of the roots with respect to changes in the coefficient, a most important consideration. Examples which have been used by the reviewer are given as illustrations. Chapter 6 covers the localization of roots by the rule of Descartes and by Sturm sequences. It also includes a most welcome elementary exposition of the Routh and Hurwitz stability criteria.

The book concludes with chapters covering the methods which are most commonly used in practice, those of Aitken–Bernoulli, Graeffe and the most useful of the iterative methods. The latter include the methods of Lin, Newton, Bairstow, Laguerre and Muller; the last of these is rather surprisingly classed as being of third order. The numerical examples presented in connection with the iterative methods are the most difficult of those given in the book. In a brief assessment of the various methods, the author declares himself in favour of the iterative methods when an automatic computer is available, and with this opinion I agree.

This is probably the most comprehensive book which is available on polynomial equations. I have for long taken the view that in spite of the important position occupied by the Fundamental Theorem of Algebra in mathematics, the practical problem of finding the zeros of polynomials is not very profound. The book therefore loses very little in presenting the subject-matter throughout in the most elementary terms possible, though some readers may like to supplement the material given here with some such work as Ostrowski's *Solution of Equations and Systems of Equations*.

Since the most important problem arising in practice is that of the condition of polynomials, I would have welcomed a discussion of the relevance of this to the accuracy attainable with the various methods. This is particularly true of deflation which is used in connection with most of the iterative processes. Another omission is the Quotient-Difference algorithms of Rutishauser, an assessment of which would have been very valuable. However, these are minor criticisms of a book which I am sure will prove popular with numerical analysts. J. H. WILKINSON.