List processing facilities in Atlas Autocode

J. S. Rohl and G. Cordingley*

* *Department of Computer Science, The University, Manchester 13*

Education is becoming a major part of the computer science scene. This paper describes the list processing facilities formally embedded in Atlas Autocode for use in teaching undergraduates.

(Received March 1969)

As computer science continues to expand as an academic discipline, it becomes more and more important to refine our ideas about programming topics, such as list processing, syntax analysis, simulation and so on, in such a way as to facilitate their teaching. In particular, if programming is regarded as a practical subject, then facilities must be provided for students to get experience in the field. It is for this reason that the list processing facilities described here have been implemented at Manchester.

General considerations

List processing facilities can be provided in three ways.

1. Implementing one of the Classical Systems such as LISP (McCarthy, 1960). This solution has the advantage that the systems are well documented, are universally understood and have been implemented widely. On the other hand, they are complete programming systems so that the student has to come to grips not only with the list processing concepts but also with all the other ideas contained in the whole system. This is not to say that those systems are not of interest, but that in the limited time available in any undergraduate course, the major effort must be devoted to the essential list processing concepts.

2. Providing a package of procedures to be used with a general purpose language such as ALGOL (for example Barnes, 1965, Trundle, 1966, and Townsend, 1968). This solution partially overcomes the major disadvantage of the classical systems. Nevertheless, it suffers from other disadvantages. Firstly, such systems do not deal with a *list* concept but with *integers* used to represent lists. Thus it is impossible to refer explicitly to lists, and rather inconvenient to refer to atoms. Secondly, these packages impose a system on the users to establish which *integers* are to be regarded as lists to aid garbage collection. Thirdly, they are very inefficient. It is true that the amount of time used on any particular machine will be quite small and inefficiencies will be of little practical consequence, nevertheless, it seems educationally unsound to use as teaching devices, systems which are for no good reason so inefficient.

3. Formally embedding list processing facilities in a general purpose language, such as ALGOL. This approach is often used for discussion of list processing (for example, see Foster, 1967) but it is less often implemented. It is one such system that is described here.

Regardless of the method of implementation, there are basically two forms of list processing that may be provided. Either the system may be concerned with explicitly manipulating storage (as in SLIP, IPL V) or it may be concerned with lists as data objects whose storage is of incidental interest (LISP). The Atlas Autocode (AA) facilities are modelled on this latter pattern partly because automatic garbage collection can be integrated with the dynamic storage allocation mechanism, and partly because it illustrates to students that types other than *integer*, *real*, *complex* and *Boolean* exist and can be formally manipulated.

The form of lists

List processing is naturally concerned with lists. In AA, a list is either an atom (an indivisible object from a list processing point of view, see below), or a string of elements separated by commas and enclosed in brackets, where an element is either an atom or another list. Formally:

⟨list⟩ ::= ⟨element⟩
⟨element⟩ ::= ⟨atom⟩|(⟨element list⟩)|( )
⟨element list⟩ ::= ⟨element⟩|⟨element list⟩,⟨element⟩

For example:

Atlas
(*Man Utd*, *Man City*, *Liverpool*, *Everton*)
(+, (*,*X*,*Y*), (*,*X*,*Y*))
( )

This last is an empty list.

List processing is concerned with the manipulation of these lists of atoms. For example, the third list above might be rearranged as:

(*,2,*X*,*Y*)

There are two forms of atom:

1. a numeric atom (at present integral);
2. a symbolic atom. This consists of any string of symbols except one containing all digits.

The list processing facilities

The facilities are built round the concept of variables of type **list**, these variables being introduced by means of a declaration of the form:

list $l1, l2, l3, l4, l5, l6$.

They are initially set to empty lists. List processing is performed by means of a list processing assignment statement in which a list variable on the left-hand side of an assignment is set to the value of the list expression on the right-hand side.

⟨list assignment⟩ ::= ⟨list variable⟩ = ⟨list expression⟩

To describe the facilities we assume the above declarations and that $l1$ has the value

$$(+,(*,,XY),(*,X,Y))$$

Formally, a list expression may be defined:

⟨list expression⟩ ::= ⟨list variable⟩
| '⟨atom⟩'
| ⟨list function⟩
⟨actual parameter part⟩
| ⟨explicit list⟩

The simplest form of a list expression is a list variable so that $l5 = l1$ gives $l5$ the value $(+,(*,X,Y),(*,X,Y))$. A list expression may also consist of an atom enclosed in quotes. Thus $l5 =$ '$l1$' sets the value of $l5$ to the atom $l1$ (i.e. the atom consisting of two symbols 'l' and '1'). The built-in list function *head* (corresponding to LISP's *car*) has as its value the first element of the list which is its parameter. Thus the assignment:

$$l2 = head(l1)$$

gives $l2$ as its value the atom $+$. There is no head to an atom or an empty list.

The associated function *tail* (LISP's *cdr*) has as its value the rest of the list which is its parameter, after the first element has been deleted. Thus

$$l5 = tail(l1)$$

gives $l5$ the value $((*,X,Y),(*,X,Y))$, and

$$l3 = tail(head(tail(l1)))$$

sets $l3$ to (X,Y).

The tail of a list of one element is an empty list; an empty list or an atom has no tail at all.

Lists may be built up from other list elements by means of the function *join* (in LISP *cons*) which inserts the list element which is its first parameter, into the front of the list which is its second parameter. Thus if $l2$ and $l5$ have the values above, then

$$l5 = join(l2, l5)$$

sets $l5$ to the original value of $l1$. The second parameter must not be an atom, though the first might well be. Lists of arbitrary complexity may be built up by means of join statements. Thus $l1$ could have been set up initially, assuming $l6$ to be an empty list, by means of the statement:

$$l1 = join('+',join(join('*',join('X',join('Y',l6))),join(join('*',join('X',join('Y',l6))),l6)))$$

This is clearly an inelegant statement whose meaning is not immediately clear. The fourth form of list expression is an explicit list. This has the same form as the data lists except that atoms are enclosed in quotes. Thus the above instruction may be more concisely expressed as:

$$l1 = ('+',('*','X','Y'),('*','X','Y'))$$

The elements may be list variables, so that $l1$ may also have been set up by:

$$l5 = ('*', 'X', 'Y')$$
$$l1 = ('+', l5, l5)*$$

AA has no type **Boolean**: the concept of a condition is used instead. A simple condition is defined:

⟨simple cond⟩ ::= ⟨expr⟩⟨comparator⟩⟨expr⟩
⟨comparator⟩ ::= $= | \neq | \geqslant | < | > | \leqslant$

where ⟨expr⟩ is an arithmetic expression. Simple conditions may be combined with **and**'s and **or**'s to form conditions. Statements are made conditional by preceding them by:

if ⟨condition⟩ **then**
unless ⟨condition⟩ **then**

or following them by:

if ⟨condition⟩
unless ⟨condition⟩

The statement is obeyed if (or unless) the condition is satisfied.

The form of simple condition is extended to include the list processing conditions:

⟨list expr⟩ **is an atom**
⟨list expr⟩ **is a number**
⟨list expr⟩ **is empty**
⟨list expr⟩ **is equal to** ⟨list expr⟩

The first three alternatives are self explanatory. The fourth condition is satisfied if the two list expressions have values which are both atomic and equal.

Transfer functions

One of the advantages of embedding a list processing system in a high level language is that the full powers of that language are available to the list processor. If use is to be made of the computation facilities then some transfer functions must be available to convert variables of type **list** to variables of other types. At present only two transfer functions are available, and these are associated with numerical atoms. The function *value* operating on a list expression whose value is a numerical atom, produces an integral result with the same value as the atom. Clearly only a numerical atom has such a value. The reverse transfer function is *newatom*. This function produces as its result a numerical atom with the same value as its integer parameter. Thus if $l1 = (2)$ then $i = value\ (head(l1))$ sets $i = 2$ and $l2 = newatom\ (i + 1)$ sets $l2 = 3$.

Routines and functions

The routine and function facilities (the procedures and type procedures of ALGOL) have been expanded to allow operations on lists. AA has, in the main, two types of parameters, those called by value (for example, **integer** and **real**) and those called by reference or simple name (for example, **integername** and **realname**). The

* The alternative would produce a different list structure from that produced by the other two (see later), though such a difference is undetectable with the facilities provided.

characteristics of these two types of parameter can be summarised: within the routine a value parameter behaves as a local variable which is set on entry to the value of the actual parameter which may be an expression. Operations on the variable within the routine have no effect on the actual parameter. On the other hand a name parameter acts as an indirect address to the actual parameter which must be the name of a variable. All references to the parameter are indirect references to the actual parameter which may therefore be altered. Two further parameters **list** and **listname** having similar characteristic properties have been added.

Two routines are permanently provided for input and output of lists. They have the following specifications:

routine spec *read list* (**listname** *l*)
routine spec *print list* (**list** *l*)

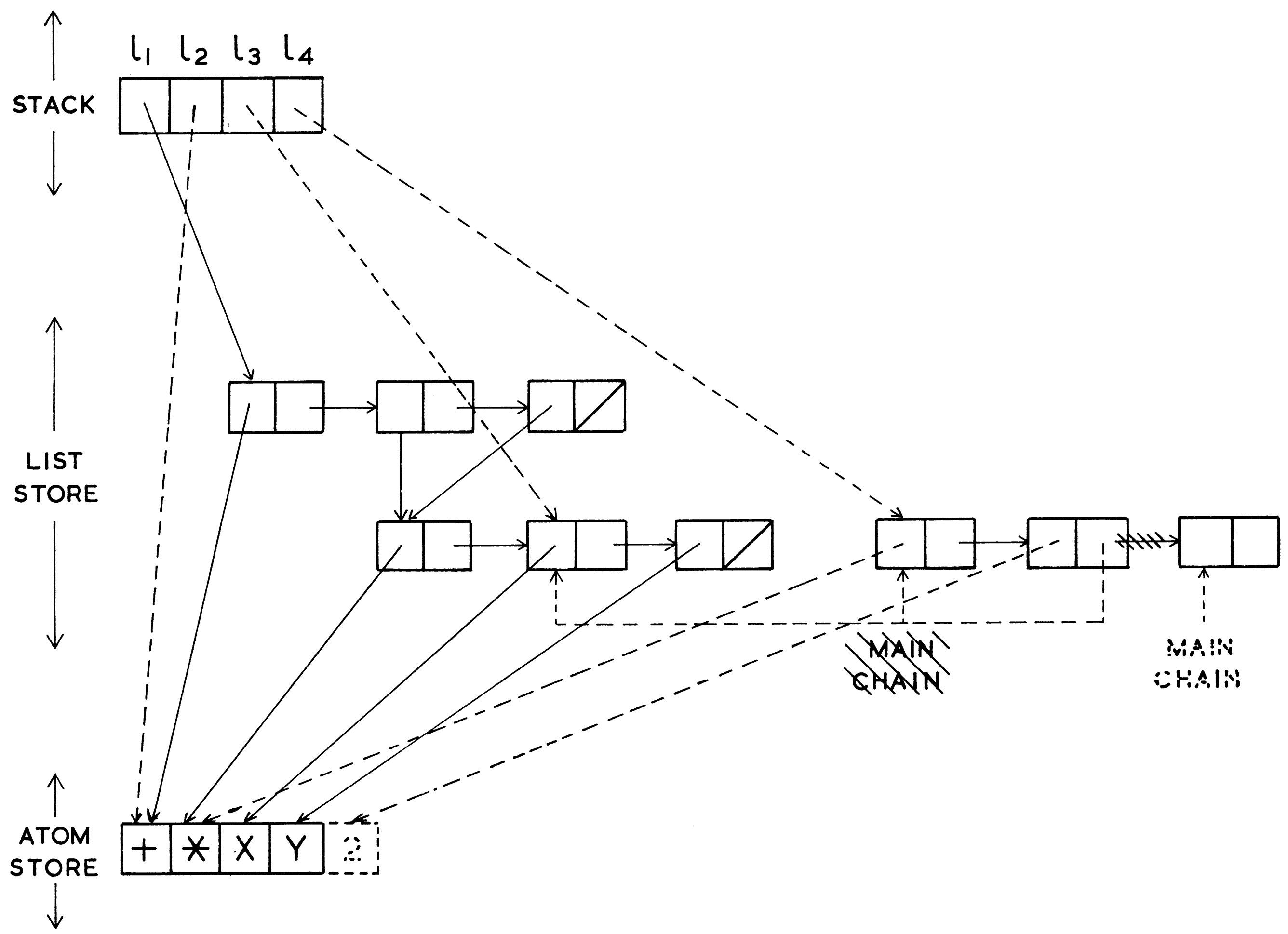
An example

To illustrate the facilities we give the classical example of analytical differentiation. We represent the sum, product or power of two quantities as a list of three elements giving the sum product or power in Forward Polish form. Thus:

$a + b$	is represented as	$(+,a,b)$
$a + bx$	is represented as	$(+,a,(*,b,x))$
$ax^2 + b$	is represented as	$(+,(*,a,(\uparrow,x,2)),b)$

We will assume for purposes of illustration that the power is always an explicit integer > 0. The following list function differentiates such an expression (e) with respect to a specified variable (x).

```
list fn diff (list e, x)
list l1, l2, l3
if e is equal to x then result = '1'
if e is an atom   then result = '0'
l1 = head(e)
l2 = head(tail(e))
l3 = head(tail(tail(e)))
if l1 is equal to '+' then
        result = ('+',diff(l2,x),diff(l3,x))
```



The store layout, in full lines of the list
***l*1=(+, (*,*X*,*Y*), (*,*X*,*Y*,))**
and, in dotted lines, after the instructions
***l*2=*head* (*l*1)**
***l*3=*tail* (*head* (*tail* (*l*1)))**
***l*4 =*join*('*', *join* ('2', *l*3))**

Fig. 1. Example of store layout

```
if l1 is equal to '*' then
        result = ('+',('*',l2,diff(l3,x)),('*',diff(2l,x),l3))
if l1 is equal to '↑' then
        result = ('*',('*',l3,('↑',l2,newatom(value(l3)
            −1))),diff(l2,x))
end
```

Implementation

The list variables like all variables are stored in the stack, and so the space they occupy is automatically deleted on exit from a routine or function. Each contains the address of the list structure which contains its value. The lists themselves are stored as list structures which are implemented along classical lines, with each element of a list occupying two (half) words, the first containing the information, the second the link to the next element of the list. The information word may be either the untagged* address of the sub-list, or the tagged address of an atom.

The link in the last element of a list is also tagged, and is, for historical reasons, the address of the atom *NIL*.

The amount of store to be used for the lists is declared by the user by means of a statement of the form:

list store 500

This initially chains up that amount of store and sets B89 to the address of its first element. This is referred to as the main chain.

The atoms are stored in a linear area of store called the atom list. Numerical atoms occupy one half-word which contains the tagged integral value of the atom. Non-numeric atoms are packed up three characters to a half-word, preceded by an untagged half-word containing the number of characters. Atoms are created in three ways:

1. During translation of a program by its appearance between quotes.
2. During running by reading a list.
3. During running by using *newatom*.

Regardless of how they are created atoms are compared with all the atoms currently in the atom list, being added to the list only if not already there. The atom list is preloaded with the atom *NIL*.

List variables, like other variables, are stored in the stack, and contain the addresses of the lists assigned to them.

Fig. 1 shows, in full lines, one possible layout of the store containing the single list $l1$ whose value is

$$(+,(*,X,Y),(*,X,Y))$$

and in dotted lines the result of the instructions

$$l2 = head(l1)$$
$$l3 = tail(head(tail(l1)))$$
$$l4 = join('*', join('2', l3))$$

As this description implies, the basic list processing operations are simply implemented. An assignment such as $l1 = l2$ is compiled into orders to evaluate the right-hand side $l2$ in the list accumulator, and an order to store the list accumulator in $l1$. If the right-hand side is simply a list variable or an atom, then only one order is required to load the list accumulator.

If the right-hand side is of the form *head* ($l2$) or *tail* ($l2$) then $l2$ is evaluated in the list accumulator and one order is planted to replace it by either its information word (for *head*) or link word (for *tail*).

As indicated earlier, only a non-empty list has a head or a tail. This condition can be checked by two orders placed before each order which evaluates a head or a tail:

Set a characteristic number in a register for use by the fault monitoring routine.
Jump to the fault monitoring routine unless the list accumulator is tagged.

Clearly to compile these for every head and tail would be extravagant since they treble the time taken for each basic operation and in any case most loops (iterative or recursive) terminate on a similar explicit test. Instead they are compiled at the option of the user. The declaration

compile atom check,

which may appear anywhere, causes the checking instructions to be compiled until it is countermanded by the statement

stop atom check.

The list expression *join*($l1$, $l2$) is compiled into a longer sequence since it takes a word from the main chain, called the join word, and a check must be inserted to test whether the main chain is empty. If it is garbage collection is initiated. The two parameters are then evaluated and stored in the information word and link word respectively of the join word taken from the main chain. Since either parameter may itself contain a join and since this join may initiate garbage collection, join words are immediately stacked after being taken from the main chain.

The second parameter may not be an atom, and if the expression is being compiled after **compile atom check** orders are planted to check for this.

The conditions all load the ⟨list expr⟩ under test into the list accumulator before testing them. To test whether the list is an atom involves one further order to test whether the list accumulator is tagged. Note that by this definition an empty list is an atom—the atom *NIL*. Since '*NIL*' is preloaded, its address is known and two orders suffice to test whether the list accumulator contains an empty list. Two further orders test whether a list is a number—one replaces the list accumulator with the actual atom, the other tests whether this is tagged.

Testing the equality of two ⟨list expr⟩'s is a little more involved, since the ⟨list expr⟩'s must be evaluated in the list accumulator, and tested both for equality and for being atoms (unless one is a literal atom).

Garbage collection

This follows classical lines in that when a word is required from the main chain which has become empty, garbage collection is automatically invoked. The lists currently referenced by list variables are scanned and

* The Atlas addressing structure allows addresses to refer down to a character. When the address is of a half-word, the bottom two lists are ignored; when of a full word, the bottom three. These bits may then be used as tag bits. We use the bottom bit throughout as a tag since it can be tested with a single order.

their information words tagged. (Provision is made for scanning common lists only once.) The whole list store is then scanned linearly returning untagged words to the main chain and untagging tagged words. These facilities have been added to the original Compiler Compiler version of AA which, unlike the hand-coded AB version, does not retain name lists at run time. Therefore, the list variables are distinguished in the stack by being specially tagged. Since they can only be found by scanning the stack, garbage collection could be slow if the program used large arrays—though this is, of course, unlikely. A better solution would have been to retain the name lists at run time.

Conclusions

These facilities are meant to give experience to undergraduate students and should be reviewed in that light. Nevertheless, it is possible to write programs to solve real problems with reasonable efficiency.

It should be noted that there are no facilities for altering list structures in any way (except during garbage collection). This is not because it is difficult to allow it—a relaxation to allow an implicit list name on the left-hand side of an assignment statement (for example, *tail*(l) = l) and as an actual parameter for a **listname** parameter (*read list*(*head*(*tail*(l)))), would enable list structures of any complexity to be created. Rather it is because we wish to retain the basic characteristic of value parameters—that any operation on a value parameter inside a routine cannot affect the actual parameter.

Many improvements suggest themselves to make this a more useful tool. Transfer functions could be provided between non-numeric atoms and variables of type **string**.* Further it might be desirable to associate values with non-numeric atoms to be used after the list processing has been done. This could be done in a number of ways, perhaps by associating non-numeric atoms with scalar variables of the same name.

Acknowledgements

We should like to thank the S.R.C. who provided the grant to enable one of us (G.C.) to pursue this work.

* There are, as yet, no string facilities in AA, so this suggestion is non-trivial.

References

Barnes, J. G. P. (1965). A KDF9 Algol list processing scheme, *The Computer Journal*, Vol. 8, p. 113.
Foster, J. M. (1967). *List Processing*, MacDonald: London.
McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine: Part 1, *CACM*, Vol. 3, p. 184.
Townsend, H. R. A. (1968). A List processing system for the 903 computer, *The Computer Bulletin*, Vol. 13, p. 120.
Trundle, R. W. L. (1966). LITHP—an ALGOL list processor, *The Computer Journal*, Vol. 9, p. 167.

Book review

An Introduction to the Approximation of Functions, by Theodore J. Rivlin, 1969; 150 pages. (*Blaisdell Publishing Co.*, \$7.50.)

This book belongs to the Blaisdell series in Numerical Analysis and Computer Science, whose jacket motif is a reel of magnetic tape—perhaps a little misleading in this case. A brief introductory chapter is followed by five others: the first three deal respectively with uniform, least-squares and least-first-power (the author's term) approximation. Chapter 4 covers polynomial and spline interpolation and the final chapter is on approximation and interpolation by rational functions. A very happy choice of material. Many will be especially grateful for the inclusion of the chapter on least-first-power (i.e. best L_1) approximation and the chapter dealing with splines.

The author begins with the definition of a normed linear space, which he uses to prove the existence of best approximations. Thereafter, Dr Rivlin commendably restricts his attention to *polynomial* approximation and, as he notes in his preface, resists making the easy extension to arbitrary Chebyshev systems. To borrow the nomenclature of P. J. Davis, this is good news for earth-men; space-men can easily supply the generalisations for themselves. Both groups will learn much from this book.

The reviewer is very impressed. Dr Rivlin has written a very fine, scholarly text which will, without doubt, be welcomed by all students and teachers of approximation theory. The material is most skilfully developed in an exciting way. What appear to be gaps in the text turn out to be filled neatly at the chapter's end by appropriate exercises.

As for misprints etc., the reviewer has only three to point out: the exponent 2 is omitted twice on page 29; a factor $w(x)$ is omitted from one integral on page 49; lastly, the footnote on page 50 seems a little out of step with the text, due to the statement of orthonormality.

G. M. Phillips (St Andrews)