Atoms and Lists

By P. M. Woodward and D. P. Jenkins

This article describes the recent work, on list processing, of J. McCarthy at MIT in a form which will, it is hoped, be intelligible to the non-specialist.

Introduction

In the past the main purpose of a digital computer has been to do arithmetic: apart from the program, the items of information in store were numbers. A machine, however, has no means of knowing whether stored items are numbers or not. They are numbers only to the extent that we treat them as numbers by performing arithmetic operations on them. Now that computers are being increasingly applied to non-numerical problems, we have to consider whether any changes are needed either in machine design or in programming methods. If the items to be processed are not numbers, operations like floating-point multiplication are inappropriate. What are the basic operations for non-numerical work? What are the best ways of arranging the information in store? There is little enough to go on, and at first we would seem to be lost in vagueness. The purpose of this article is to describe the recent work of McCarthy (1960), whose programming system called LISP seems to answer some of our questions. McCarthy's answer may not be the only one, but it is a major step in the right direction.

Atoms and Lists

If the items of information are not numbers, they may be yes's and no's, points or lines, names of people, algebraic symbols, elements of a group, musical notes or anything. McCarthy originally called them "objects," but has changed this to *atoms*. An atom is any irreducible item of information, or any item which the programmer wants to treat as irreducible. In algebraic work, which is one of the obvious applications for LISP, typical atoms might be X, +, COS, 7 or (. So far, generality is complete. We now have to ask ourselves what we want to do with our atoms, to which we can only say that we want to be able to group and re-group them in any way we please. For example, integer arithmetic is merely a grouping and re-grouping of the atoms 0 to 9. Removal of brackets in algebra, formal differentiation, sorting, and so on, can all be included in this broad statement. Before we can get really started, however, it is necessary to decide on a structure for a group of atoms, and it will be this structure which will determine the character and degree of generality of the mathematics. LISP is based on *list structures*. A *list* is defined as an ordered array of atoms or lists, separated by commas and enclosed in brackets. For example,

$$x = (A, B, C) \qquad (1)$$

is a list, where A, B, and C are atoms. So too is

$$y = (A, ((B, C), D, (E, F)), G, H) \qquad (2)$$

It is the recurrence of the term "list" in the definition of a list which gives these arrays their special character.

Atoms need not be restricted to single block letters, and a useful plan is to allow any succession of capitals or digits. Thus, SIN, COS, PLUS, A3, and 27 would all be allowed as atoms.

S-Language and Meta-Language

In ordinary mathematical work, we are accustomed to the use of two distinct languages which work at different levels of reality: namely arithmetic and algebra. There is never any confusion, because we use a different alphabet for each. We know that 23 is a *number* and that x, being a letter, is an "unknown quantity" which *stands* for a number. In LISP, there is need for a similar distinction, so we reserve upper case for atoms and lower case for symbols which stand for atoms or lists. The letter x on the left-hand side of (1) is standing for a list, and we can say that x in this example has the *value* (A, B, C). Once this has been said, it is important to realize that no further evaluation is possible. Whilst letters like x have the same generality as algebraic variables, standing for different things in different problems, letters like A, B, and C do not stand for anything, and are already quite particular. When writing actual atoms or lists in upper-case letters, we are using what McCarthy has called S-language, and expressions like A or (A, B, C) are called S-expressions. This term is short for Symbolic Expressions—an unfortunate description in view of the dictionary definition of a symbol as something which stands for something else, but one which is intended to emphasize that LISP is entirely concerned with the manipulation of abstract entities.

The language of x and y is called M-language or meta-language, and it must be extended to provide a notation for functions of lists. A function of a list has as its value another list, or perhaps an isolated atom. In other words, the evaluation of a function of an S-expression gives another S-expression. Later in this article, for example, we discuss the function $ff[x]$ which represents the operation of finding the first atom in any list x. In meta-language, all letters are printed in lower case, brackets are square and semi-colons are used as separators, to avoid overlapping with the set of characters employed in S-expressions.

A program in LISP can always be thought of as a problem in function evaluation. A general process is described in meta-language by defining a function of one or more lists; particular values for these lists are then given in S-language, and the job of the program is

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

to evaluate the function and give an answer in S-language. It will immediately be seen that we have here an exact parallel with autocoded numerical work, where the general process is described in algebra, particular starting values are given numerically, and the final answer is also numerical.

Machine Format

To achieve a programming system, we have to decide how lists can be stored in a computer so as to preserve their nesting structure. Here there appears to be considerable choice, and room for experiment. McCarthy's arrangement is fundamental to LISP and is illustrated in Fig. 1 for the lists (1) and (2) above. Each oblong represents a word in the computer store, partitioned down the middle. A box drawn empty contains the address of the word to which its outgoing arrow is pointing. All other boxes can be supposed to contain the atoms written in them, though actually they contain the addresses of these atoms. A box with a diagonal line contains the special atom NIL which marks the end of a list.

To illustrate more fully this way of storing lists, the following is a possible allocation of three stores for the list (A, B, C) shown in Fig. 1(*a*):

5)	C	NIL
16)	A	28
28)	B	5

The first column gives the address of the stored computer word, the second and third give the items stored in the two halves of that word. The addresses 5, 16, and 28 are here chosen quite arbitrarily to show that all notions of sequential storage have gone by the board. Reference to the above list is, of course, via store 16 where the list starts, and the computer must be made to keep a note of this number. Any list processing scheme must, however, disguise all these details of storage allocation from the user.

One reason for abandoning sequential addressing of data is that it is unsuitable for branched lists such as the one shown in Fig. 1(*b*). Another is that lists are variable and unpredictable in length, unlike the numbers, vectors,

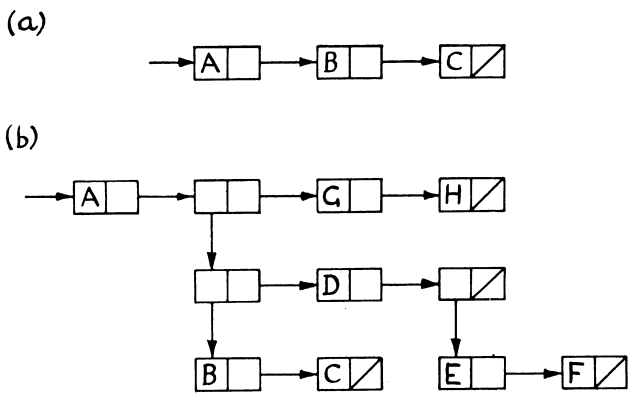

Fig. 1.—Machine format for lists. The capital letters are atoms

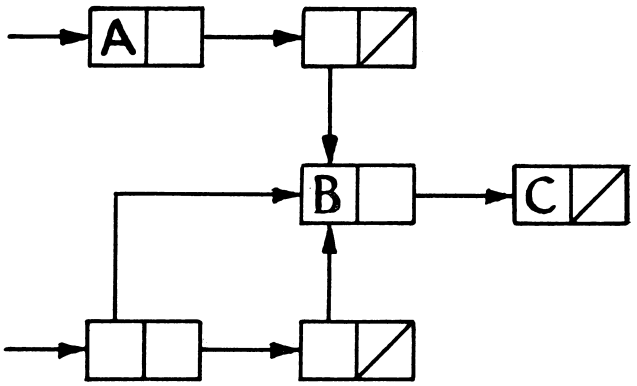

Fig. 2.—Sharing of a sub-list

or matrices in ordinary computer work. The organization required to store branching lists of varying length sequentially would be far greater than is required in LISP, which merely seizes on any vacant store which comes to hand. More will be said of this later, but it is important to realize that the concept of almost random storage is really one of the main ideas behind LISP: it is what makes the partitioning of every word into halves necessary.

To give a further example, we show below a possible arrangement for storing the list *y* shown in Fig. 1(*b*). To avoid needless confusion to the reader, we have here replaced the haphazard addressing which would occur in a machine by an orderly arrangement. The addresses 1 to 11 can be thought of as standing for any eleven different integers.

1)	A	2
2)	5	3
3)	G	4
4)	H	NIL
5)	8	6
6)	D	7
7)	10	NIL
8)	B	9
9)	C	NIL
10)	E	11
11)	F	NIL

The particular way in which the linking of elements is done in LISP is, at first sight, rather restrictive. For example, it is easy to trace through a list from head to tail, but very difficult to do the reverse. In fact, this turns out to cause surprisingly little embarrassment and does not in any way impair logical generality. (LISP is a "universal" system, capable of computing anything computable.) In one major respect, the lack of backward address references is a real virtue, since it enables two lists to share a common sub-list. For example, the lists

(A, (B, C)) and ((B, C), (B, C))

can be represented as in Fig. 2, which shows a double economy.

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

Dot Notation

When we want a really direct notation for machine format, we can either use boxes as above or—more conveniently—an expanded form of the comma and bracket notation, using dots.

In dot notation, the list

(A, B, C)

becomes

(A.(B. (C. NIL)))

The dot is a precise representation of the bar which subdivides the two halves of a box in box-notation. It connects two atoms or lists and only two. As a further example, the list (2) shown in Fig. 1(*b*) becomes

(A.(((B.(C.NIL)).(D.((E.(F.NIL)).NIL))).
(G.(H.NIL))))

It will immediately be obvious that the comma notation is more compact than this literal representation of machine format. Dot notation is, however, helpful for understanding LISP properly, and moreover it does not force the use of NIL. For example,

(A.B)

is a perfectly acceptable list which cannot be expressed in comma notation.

Dot notation and comma notation can be mixed without ambiguity, and this brings out their differences. In *pure* dot notation, round brackets are used to enclose every connected pair of atoms or lists, and are *only* used in this way. Thus (A) would be meaningless. In comma notation, on the other hand, (A) is a list with one element, and is (A.NIL) in dot notation. Since the elements of lists can be atoms *or lists*, it follows that ((A)) or (((A))) are allowed expressions, and their machine format is shown in Fig. 3. As an example of mixed notation, we could have the S-expression

((A.B), C, D, E)

which means

((A.B).(C.(D.(E.NIL))))

shown also in machine format in Fig. 4. The preoccupation with notation may seem burdensome, but we must remember that notation is, so to speak, the title of the paper.

The Primitives

We are now equipped to start operating on lists in meta-language. Just as the whole of algebra can be built up from a few primitive operations on numbers (like addition), so we can build up a complete information-processing scheme for list structures from half a dozen basic operations. These have the strange sounding names

atom, *eq*, *car*, *cdr*, *cons* and *cond*.

Each of these can be regarded as a function (like *sin* or *log*), whose arguments are placed in meta-linguistic square brackets. Where there is more than one argument to a function, semi-colons are used as separators.

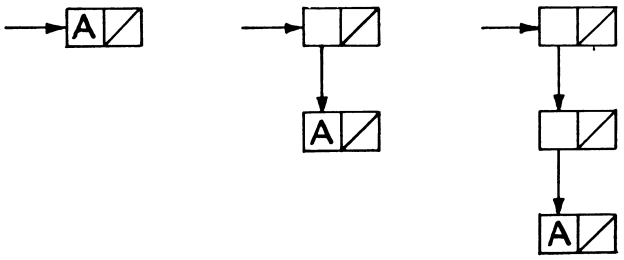

Fig. 3.—(A) and ((A)) and (((A))) in machine format

atom [*x*] is a Boolean function (i.e. logical proposition) whose value is either true or false. The atoms T and F are used for the answer, which is T if *x* is an atom and F if it is not.

eq[*x*;*y*] is also Boolean. Its value is T if *x* and *y* are equal atoms, F if *x* and *y* are different atoms, and undefined if *x* or *y* are molecular.

car[(*x*.*y*)] is the first member of the dotted pair: *x*. Here *x* and *y* may be atomic or molecular, i.e. any S-expressions.

cdr[(*x*.*y*)] is the second of the pair: *y*. Again *x* and *y* are any S-expressions.

cons[*x*;*y*] has the value (*x*.*y*) where *x* and *y* are any S-expressions.

cond[[p_1; e_1]; [p_2; e_2]; etc.] is the all-important *conditional* function. Here the arguments must be written in pairs. The *p*'s are logical propositions, which may (like *eq* above) be defined or undefined. If defined, they must evaluate to T or F, whereas the *e*'s stand for *any* S-expressions. The value of *cond* is found by reading the arguments from left to right and taking the *e* which follows the first *p* whose value is T. If no *p* is true, or if an undefined *p* is encountered before a true *p*, the value of *cond* is undefined. A shorthand notation for *cond* is

$$[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \text{etc.}]$$

with the function name left out.

The Boolean functions *atom* and *eq* can be used as propositions in *cond* functions, and these in turn can

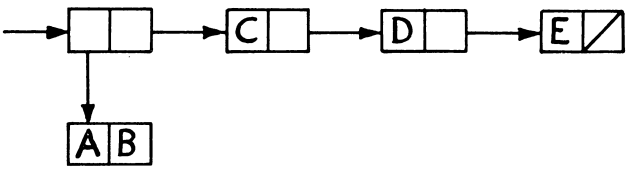

Fig. 4.—List which cannot be represented in pure comma-notation

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

supply us with more Boolean functions. *Car* and *cdr* are used for splitting a list into pieces, and *cons* is used for constructing new lists from given pieces. From these six functions, all computing operations can be built up.

It should be pointed out that the choice of names for functions is quite arbitrary, and whilst most of the above are obvious abbreviations, *car* and *cdr* are hardly likely to appeal in the long run. (They were chosen by McCarthy because of their significance in relation to a particular computer.) The present writers have used "fore" and "aft" for lecture purposes, without seriously proposing their adoption. LISP programmers soon find their literary imaginations heavily taxed in the search for more and more function names as a library of routines is built up.

A Simple LISP Program

An elementary problem in programming will help to clarify these ideas. Take the following:

"Given any S-expression x, find the first atom occurring in it."

First, let us make up a function name for the operation which finds the first atom in x, say $ff[x]$. A LISP program for $ff[x]$ is merely a formula which expresses ff in terms of previously defined functions. *The formula can be recursive*: it is permitted that ff should occur on both sides of the equation. Thus,

$$ff[x] = [atom[x] \rightarrow x; \mathrm{T} \rightarrow ff[car[x]]]$$

is a LISP program for evaluating $ff[x]$. In words, it means

If x is an atom, the value oi $ff[x]$ is x.

Otherwise the answer is $ff[car[x]]$.

The use of T needs explaining. There are two propositions in the *cond* function on the right, viz.

(1) $atom[x]$

(2) T

Since $atom\ [x]$ is a function of x, it cannot be evaluated until x has been evaluated or specified in S-language. (This is like saying we cannot compute $sin\ x$ until x is given numerically.) But it happens in this problem that if $atom\ [x]$ has the value F, we want to evaluate $ff[car[x]]$ whatever the value of x. The second proposition in the *cond* is therefore a *known* quantity (like a number) and can be written directly in S-language as T. This will ensure that the second proposition is always accepted when the first is rejected. The program for ff thus employs a mixture of meta-language and S-language, analogous to that of letters and numbers in algebra.

The way in which the program actually works can be followed by means of an evaluation, say for

$$x = (((\mathrm{C}.\mathrm{NIL}).\ (\mathrm{B}.\mathrm{NIL})).\mathrm{NIL})$$

Clearly $ff[x]$ is here C. We arrive at this result by first checking that x is not just an isolated atom. If it were,

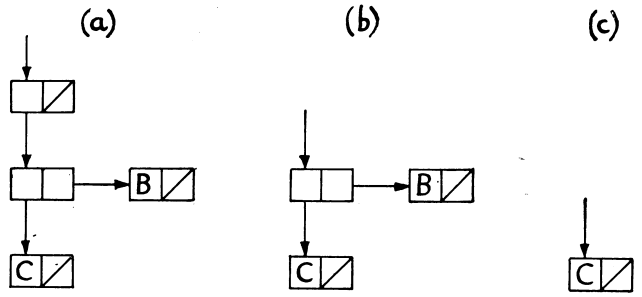

Fig. 5.—Steps in the evaluation of *ff*, see text

x would be the answer. But if not, we look at $car[x]$, which is the first box in Fig. 5(*a*). This is (in full)

$$((\mathrm{C}.\mathrm{NIL}).\ (\mathrm{B}.\mathrm{NIL}))$$

as shown in Fig. 5(*b*). Again apply ff. Again the expression is not an atom, so we take the *car* again:

$$(\mathrm{C}.\mathrm{NIL})$$

as in Fig. 5(*c*). Again we apply ff, again the expression is not an atom, so take the *car* of it which is

$$\mathrm{C}$$

Taking ff once more, we find at last that the argument is an atom, and our formula tells us that, when this happens, the atom is the answer. In more involved recursions, it is necessary not only to "unwind" the recursion to its end, but also to wind it up again before the answer can be obtained. This is a distinction which need not trouble us in an introduction to the subject, but needs attention by the programmer who wants to avoid lengthy cycles of computation on the machine.

It should be noticed that recursive definitions are the only way of achieving loops in the pure form of LISP, and that a LISP program is all rolled into one equation. The beginner finds that a new mode of thought is necessary in this kind of programming, a mode which in certain types of work may be the most natural.

How LISP Programs are run

A machine must obey its own code. If a foreign code is to be imposed on it, it must at some stage be translated into the engineered code of the machine. This can be done either through the agency of a translation program which converts the whole program into machine code before the actual problem starts, or it can be done by means of a translation or "interpretation" of each foreign instruction as and when it is encountered during actual running. The differences between the two methods show up in various ways. With preliminary translation, the stored program during actual running of the problem is in machine code. All foreign code has by this time been disposed of. With running interpretation, the program is stored in the foreign code throughout the problem. The machine is never allowed to obey the foreign instructions (since they would not be understood), but obeys instead a cycle of machine instructions

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

which are independent of the particular problem. These examine each foreign instruction in turn, as though it were an item of data, and the form of the foreign instruction acts like a key which brings into play an appropriate prefabricated sequence of machine instructions. All possible required sequences are stored inside the interpreter program. The simplest way to make a computer accept a problem in LISP is to provide an interpreter for the particular machine which is going to be used. This has been done on the IBM 704 at MIT, and it has now also been done by one of us (D. P. J.) on TREAC at the Royal Radar Establishment; though too recently to report detailed experience of actual problems.

To run a program, the LISP interpreter and special input and print routines must first be supplied to the computer. The problem can then be fed in from a tape bearing the LISP program and its data. For example, the information on the problem tape might be

"Evaluate $ff[x] = [atom[x] \rightarrow x;\ T \rightarrow ff[car[x]]]$

for $x = (((C.NIL).(B.NIL)).NIL)$"

as discussed in the previous section, and the printed answer which should ultimately appear would be C. In fact, problem tapes are not prepared as above, but are first converted into S-language. It is here that many readers of McCarthy's paper become confused and give up.

On the hypothetical tape shown above, the program is written in meta-language and the data for the problem (the value of x) is written in S-language, ready to be stored as a list as described earlier. But when we come to decide on the best way to store the program in the machine, we find that it too can perfectly well be stored in list form, just like the data. Since S-language is specially designed for list structures, we might as well convert the program into S-language and use the same input routine both for the program and the data. The details of how this is done are irrelevant: it is sufficient to say that we end up with a tape employing only capital letters and numerals, round brackets, commas, and dots. (Function names such as *car* are treated as atoms, and *car* is therefore punched on tape as the atom CAR.) Languages which can be expressed in the same form as the data upon which they operate are known as *proto-syntactic*, and LISP is of this type.

The program and data are not the only lists which must be present in the machine store. The result obtained by evaluating a LISP problem is only a re-grouping of the input atoms, and the letter sequences used for these atoms, which differ from problem to problem, are made up by the user. It follows that these names must be retained in store (unlike the input symbols in an algebraic autocode) throughout the problem. In LISP this is done by linking all atoms together on an "atom list" which is made up of "property lists," one for each atom. Each property list contains information like the sequence of letters used at input and output, and, where the atom is a function name, some reference either to a sequence of machine instructions for evaluating the function, or to a list defining the function in LISP symbols. Other lists, used as working space by the interpreter, get formed and discarded during the evaluation of an expression.

The Interpreter

It may be of interest to describe the operation of the interpreter, at least in outline, since it may suggest extensions in the logical design of computers to be used mainly in list working. The evaluation of a LISP expression is basically the evaluation of a complicated function of arguments. To evaluate any function, the interpreter first examines the property list of the atom representing it. If the function is defined by a machine-language subroutine, these instructions are obeyed, and they control re-entry to the main interpreter whenever it is necessary to evaluate an argument of the function. Since, however, most functions will be defined by other LISP expressions, it is more probable that the property list will refer the interpreter to a definition as something more to be interpreted. After noting the position of this definition, the interpreter evaluates the arguments of the function noting the results on a "parameter list." It can then return to evaluate the functions in the definition, a situation only one stage removed from that at the start of the problem and involving the same process of examining property lists and evaluating arguments—which may now depend on expressions stored in the parameter list as well as on the original arguments. So far, we have glossed over the evaluation of the individual arguments. In the general case each of these will be a function of arguments to be evaluated by the process already described, leading to private extensions of the parameter list which can be lopped off when the answer has been found. It may well be asked how an answer is ultimately found; in the end, it comes to evaluating a function represented by a machine program when all its arguments have also been evaluated.

The whole situation is like a treasure hunt in which many of the clues are only handed over after successfully completing sub-treasure hunts starting from the points where the clues are kept. Of course, any clue in a sub-treasure hunt may require yet another hunt. . . . Clearly it could become difficult to remember where to go after completing one of the internal treasure hunts. One way of dealing with this would be to make a note of each new starting point on top of an in-tray on beginning a hunt, removing the top item of the tray, which gives the reporting point, at the end of a hunt. A similar device is used by the interpreter to keep track of its wanderings, but, true to type, its in-tray need not be sequential but can take the form of a list, usually called a "push-down list." This push-down list plays an important part in the operation of the interpreter, enabling it to overcome its embarrassing property of being a recursive routine, that is one which can use itself as a subroutine. It is fundamental that this worry should not be handed on to a user writing programs in LISP, itself a recursive

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

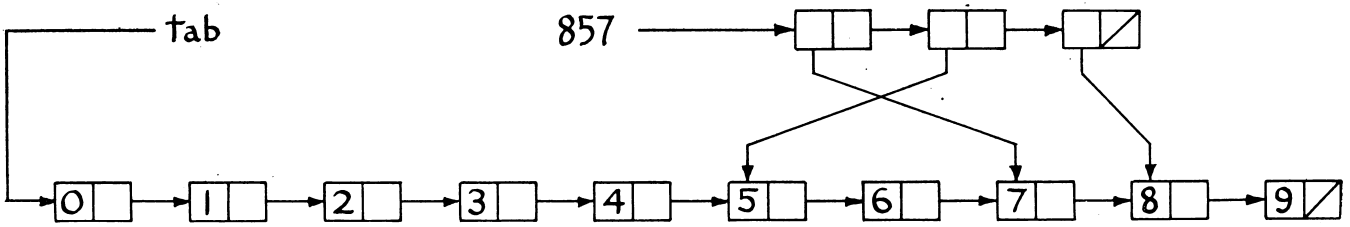

Fig. 6.—Stored form of list for an integer, using *tab*

language, and this is much easier to do by interpreting a LISP program rather than by translating it into machine code and trying to forecast the recursions that can occur without carrying out the problem.

It will have been noticed that no reference has been made to storage addresses in writing LISP expressions. In this, LISP resembles an ordinary algebraic autocode, but the two differ in that LISP does not require the user to say how many stores are to be reserved for data. Indeed from the above description it will be clear that it is nearly impossible to make any such estimate. At all stages of a LISP calculation, not only at input, the interpreter must be able to allocate stores to results obtained by *cons* operations, which generate new lists. To do this, all unused stores are linked together in a simple list called the "free list" from which they are removed when needed. When the free list is exhausted, there may still be storage space available from lists once formed as intermediate results but now discarded, like the familiar working spaces of ordinary programming. Such stores are recovered by a special part of the LISP interpreter called, in the U.S.A., the "garbage collector." This traces all lists in current use (the atom list, parameter list, push-down list and the expression being evaluated), adding to a new free list any stores which cannot be reached via an active list.

A Further Example

To show what a more complicated LISP program might look like, McCarthy gives as his example a program for algebraic differentiation. This turns out to be remarkably easy to write in LISP, and has been run on TREAC. The fact that it was possible to use McCarthy's program as published (in addition to a different local version) demonstrates the value of a machine-independent language.*

As a further example, we give now a program for finding the sum of two positive integers. Clearly this is not a practical use of LISP, but it illustrates how the basic rules of integer arithmetic can be expressed. The program works for numbers with any number of digits, and in any scale of notation. The latter is determined solely by the list for *tab*. It will be noticed that the program is not all embodied in a single equation, as implied earlier, for LISP input can be arranged to allow for separate defining equations for functions or lists appearing in the main equation, in this instance the one which has *sum* on the left.

$$sum[x; y] = [null[y] \rightarrow x; \mathrm{T} \rightarrow inc[x; y; tab]]$$

$$inc[x; y; b] = [eq[caar[y]; car[b]] \rightarrow cons[car[x]; sum[cdr[x]; cdr[y]]];\ \mathrm{T} \rightarrow inc[step[x]; y; cdr[b]]]$$

$$step[x] = [null[x] \rightarrow cons[cdr[tab]; \mathrm{NIL}];\ null[cdar[x]] \rightarrow cons[tab; step[cdr[x]]];\ \mathrm{T} \rightarrow cons[cdar[x]; cdr[x]]]$$

$$tab = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$$

Some words of explanation are needed before this program can be followed. First, there are some functions hitherto undefined, *caar*, *cdar* and *null*. Since these functions are of general use, they can be incorporated in the LISP interpreter for use by all programs. (The other new functions in this example, *sum*, *inc* and *step* are, of course, defined by the program itself.)

caar[ ] is short for *car*[*car*[ ]]

cdar[ ] is short for *cdr*[*car*[ ]]

etc.

null[*x*] is a built-in Boolean function whose value is T if *x* is the atom NIL, but otherwise F.

Secondly, it is necessary to describe the structure adopted for the lists *x* and *y* which represent the numbers to be added. The natural way of representing the integer 857 would be by the list (8, 5, 7), but for arithmetic operations it is more convenient to store it with the order of the digits reversed and to use sublists of *tab* for each of the digits. Thus 857 is represented in the machine by the list

((7, 8, 9), (5, 6, 7, 8, 9), (8, 9))

which is more clearly understood from Fig. 6. The *tab* list contains the basic notion of counting and gives the successor digit to each digit in whatever scale is used for arithmetic, here decimal. After this, the next stage is to

* The fact that TREAC took 5 seconds to prove that $dx/dx = 1$ should not be taken too seriously, since the size of high-speed store necessitates continual transferring of information to and from a magnetic drum.

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024

be able to count with carries in a many-digit number, and this is the purpose of *step*. Finally, *sum* is found by introducing a working space whose initial value is *tab* which, according to our convention, stands for zero, and the loop using this is the function *inc*[x; y; b]. To make full use of place notation, *inc* only adds one digit of y to x, so that it repeatedly calls in *sum* as well as itself, making a quite involved recursive loop. Those familiar with conventional programming might consider the problems involved in using two subroutines in the way *sum* and *inc* are used here.

Discussion

Most readers who have persevered to this point without omitting anything except the previous section will no doubt be wondering what possible applications there might be for yet another programming scheme which looks, at first sight, as unnatural as some of the more unattractive machine codes. Is LISP an intricate novelty, or should it be taken seriously? As to immediate applications, the most obvious is the problem of compilation, or autocode translation. It has been stated that the automatic formula-translation program for the IBM 704 computer, known as FORTRAN, took about twenty man-years to write in machine code. Our TREAC algebraic compiler (Pearcey *et al.*, 1960) took about three man-years, but is much simpler. Apparently what is lacking here is a suitable language in which to write down the logical operations which the machine is required to perform. It is not especially difficult to describe the translation process in words. It is virtually impossible in ordinary mathematical notation, and experience has shown that it is extremely difficult in machine code, because the primitive operations of a machine do not correspond to the larger units of ordinary logical thinking. Much experimenting is being done, particularly in the U.S.A., in an attempt to invent new languages to ease the work of the logical programmer, whose problems are not at all arithmetical in nature. LISP is one of the most mathematically appealing of these experiments, though list structures play a central part in many other systems, published and unpublished. (See, for example, the references to papers by Newell *et al.*, Perlis and Thornton, Windley, and Brooker and Morris.)

The particularly attractive feature of LISP is the way in which, for the first time, a programming scheme has been disguised as a formal mathematical language. The exceptionally heavy emphasis on recursion will, at first, seem unnatural to the programmer accustomed to conventional looping. Let a programmer attempt, however, to write a machine code program for algebraic differentiation and he will find that the mode of thought engendered by sequential instructions runs counter to the manner in which he naturally thinks about such procedures, which is implicitly recursive. For instance, to differentiate a product xy with respect to t, one thinks

$$\frac{d}{dt}(xy) = x\frac{dy}{dt} + y\frac{dx}{dt}.$$

The act of differentiation *recurs* on the right-hand side and, furthermore, if x is itself a product, an *automatic* program should note that dx/dt can then be reduced by applying the same equation over again with different parameters. All this is natural in LISP.

What applies to differentiation applies also to such problems as algebraic simplification, which are closely related to bracket manipulations required in autocode work. But beyond all the detail, it is perfectly clear that the tree-like forms of list structures are more suited to storage of certain types of data than the simple linear lists which we normally handle in conventional programming.

References

BROOKER, R. A., and MORRIS, D. (1960). "An Assembly Program for a Phrase Structure Language," *The Computer Journal*, Vol. 3, p. 168.

BROOKER, R. A., and MORRIS, D. (1961). "Some Proposals for the Realization of a Certain Assembly Program," *The Computer Journal*, Vol. 3, p. 220.

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

NEWELL, A., and SHAW, J. C. (1957). "Programming the Logic Theory Machine," *Proc. 1957 West. J.C.C.*, p. 230.

NEWELL, A., and TONGE, F. (1960). "An Introduction to Information Processing Language V," *Communications of the Assoc. Comp. Mach.*, Vol. 3, p. 205.

PEARCEY, T., HIGGINS, S. N., and WOODWARD, P. M. (1960). "The Mark 5 System of Automatic Coding for TREAC," *Annual Review in Automatic Programming*, Vol. 1, p. 23.

PERLIS, A. J., and THORNTON, C. (1960). "Symbol Manipulation by Threaded Lists," *Communications of the Assoc. Comp. Mach.*, Vol. 3, p. 195.

WINDLEY, P. F. (1960). "Trees, Forests and Rearranging," *The Computer Journal*, Vol. 3, p. 84.

Downloaded from https://academic.oup.com/comjnl/article/4/1/47/379778 by guest on 13 March 2024