LISP - AN AMICUS CURIAE BRIEF

V. R. Pratt

1/10/77

-- To the complexity of
building a single interface
between people, machines,
and problems, which has
made this brief so long.

The department is presently considering the available choices of facilities for a department-wide educational computing resource. One such facility must be a language or languages. Of all the facilities (editors, processors, mass storage media, consoles, etc), the choice of language has the greatest impact on the student, if not on the professional programmer. This is because every encounter he has with software, whether on a machine, in the class-room, or in an exam, must go through the medium of language. For example, at present the only choice of facilities critical to the department's various algorithms courses (6.046, 6.073, 6.851J and 6.854J) is that of language.

The Ad Hoc Committee on Educational Computing Resources has narrowed the choice to APL, PASCAL (or a similar algebraic block-structured language) and LISP, but appears to be unwilling to narrow the choice any further, and instead proposes to make all three available on an equal footing. The obvious democratic advantages of such a solution are counter-balanced by the increased maintenance costs associated with promising full support for a variety of

languages. Moreover, the choice of utility language for general classroom use (e.g. for expressing algorithms) will still be at the discretion of the individual instructor, requiring the students to be proficient in all languages supported for this purpose by the department. While it is reasonable to require our students to know all the department's languages by the time they graduate, it is unreasonable to expect them to know them all by the end of their first year. Nor is it reasonable to expect them to know them all equally well; in fact, given the demands we already place on our students' time, it seems unfair to demand a complete mastery of more than one language. A working knowledge of a variety of languages is without doubt a vital part of a computer science education, but we should not confuse working knowledge with complete mastery when choosing a language for a course on the basis that the student has been exposed to it at some time during his education.

This position paper is intended to supply the committee with information about LISP that can come only from someone who has used LISP extensively yet who has also had a comparable exposure to other languages competitive with LISP. In my own case I used the implementation of ALGOL due to Randell and Russell [9] from 1964 to 1969 at the Basser Computing Department of the University of Sydney, and also taught ALGOL for approximately fifty contact hours in several departmental "crash courses". My LISP experience extends from 1970 to now. It is hoped that the deeper understanding of LISP that this paper attempts to supply will be of value to the committee in determining the optimal number of languages to be given full support by the department.

There being no universally agreed on dialect of LISP to date,

I have chosen to describe MACLISP, the dialect implemented at MIT. (The major alternative dialects are INTERLISP, formerly BBN-LISP, and UCI-LISP, a derivative of Stanford's LISP 1.6.) Even with such a concrete object as an implementation there is room for interpretation of what has been implemented. Thus it must be realized that the following represents one individual's perspective on one dialect of LISP.

To a non-LISP-user, LISP's most forbidding aspect is its notation, and so it is appropriate before entering the main discussion to say a word or two about this. To many LISP users the standard notation offers advantages such as simplicity, ease of learning, and the appearance of being data (as indeed it actually is). However, those who feel comfortable with algebraic languages and do not require anything else need not be put off by the standard notation. MACLISP has an alternative extensible algebraic syntax (called "CGOL") which is similar to that of popular algebraic languages such as ALGOL and PL/I, except that, being just a notational variant of LISP, it inherits the many advantages of LISP that we document below. Item 6 of section A contains an ALGOL program together with its remarkably similar translation into this algebraic variant of LISP.

A. MERITS OF LISP.

There are ten sections below, with first sentences as follows.

1. LISP is versatile.
2. LISP is efficient.
3. LISP uses a standard character set.
4. LISP is interactive.

5. LISP is modular.
6. LISP is notation-independent.
7. LISP is applicative.
8. LISP is widely used in academia.
9. LISP is used by half the MIT Computer Science faculty.
10. No other language enjoys all the above advantages of LISP.

1. LISP is versatile. Although LISP is caricatured by non-LISP users as being of use mainly for manipulation of irregularly structured data, this caricature does little justice to the careful work done by McCarthy, Levin and others in the formative years of LISP (around 1960) in developing a mathematically clean yet general programming language. Certainly Artificial Intelligence applications were a concern during that development; after all, AI was the nutrient medium within which LISP developed. Yet the language has managed to remain remarkably free of the concessions one might expect to arise from such pressures, and is in our view one of the most domain-independent languages currently enjoying wide usage.

We may illustrate LISP's versatility by reference to its data types. These are:

numbers	integers (unlimited size), reals
bit vectors	various applications, e.g. PASCAL's sets
booleans	T and NIL (for false)
atoms	serving double duty as strings and variables
lists	for which LISP is best known
property lists	various applications, e.g. PASCAL's records
arrays	unrestricted as to type or dimension
function(al)s	using LAMBDA and APPLY

programs using EVAL and QUOTE

In the MACLISP implementation the above data types are almost "first-class citizens," a term used by the implementors of POP-2 [5] to describe a data type that can be passed as a parameter, returned by a function as a value, assigned to a variable, and tested for equality. LISP's data types include some for which equality cannot be decided, namely the last two. If we rule out the last requirement, then all LISP data types are first-class citizens.

It is hard to appreciate what first-class citizenship really means until one has programmed with and without it. A generation of LISP programmers has capitalized on this asset of LISP in order to express themselves more economically yet more clearly. The examples one finds in textbooks and manuals of LISP generally confine themselves to lists and numbers, but the same style carries over to the other data types of LISP when they are made available as first-class citizens. I for one can vouch for many occasions on which this attribute, applied to arrays, property lists, functions and programs has been of value. For some (Joe Weizenbaum for example), the first-class citizenship of the data types FUNCTION and PROGRAM truly set LISP apart from other languages. Around MIT the notion of "procedural embedding of knowledge," starting with theses by Carl Hewitt and Terry Winograd, has capitalized on this asset of LISP. A good programming style can be developed along these lines in which one stores information in small modules that can be evaluated by LISP when they need to be queried. The advantage of this style is that such information can be made context-dependent because like all LISP code it has access to the environment. Moreover a module can be "intelligent" about what it returns, possibly calling on other modules

for help before making up its mind. Carl Hewitt has built a whole programming language [2] based solely on this philosophy and has demonstrated how ACTORS (an apt term for the active modules of information that characterize this style of programming) can by themselves supply the only foundations needed for a versatile and efficient programming language, using methods analogous to the corresponding demonstration for the pure lambda-calculus. Some of my own software benefits from this style of programming, which is only possible for me as an ordinary LISP user because of LISP's according programs first-class citizen status.

2. LISP is efficient. Another myth popular among non-LISP users is that to use LISP one resigns oneself to gross inefficiency. To put this shibboleth to the test, members of the MACSYMA project took some numerical benchmark programs of the sort that one would normally think of as being well-suited to FORTRAN compilation, and compared their running times under each of an (admittedly old) FORTRAN compiler and the LISP compiler used at MIT on ITS [1]. Both compilations were performed on the same machine, a PDP-10. The LISP compiler won! With a little thought it becomes apparent that inefficient object code does not inhere in a language but rather is the result either of the program demanding something difficult such as a complicated parameter-passing task, or of the compiler-writers not doing a good job. After all, why should the FORTRAN statement

```
A(I,J) = B(I,K)*C(K,J)
```

and the LISP statement

```
(STORE (A I J) (TIMES (B I K) (C K J)))
```

produce different object code? In fact, in the experiment cited above, the slight superiority of the LISP code (involving an insignificant factor of about 1.2) was traceable not to the code

generated for the arithmetic parts of the program, which was almost identical in each case, but rather to the more efficient procedure calling in LISP. This I feel convincingly disposes of the argument that FORTRAN (and hence presumably most other high level languages) is more appropriate when efficiency is needed.

Abraham Bers' Plasma Dynamics group at MIT, which although in EECS is not a part of the Computer Science laboratories (LCS and AI), does considerable "number crunching," having used several hundred hours of computer time for a variety of heavily numerical problems. John L. Kulp of that group has experimented with a few numerical problems using FORTRAN on MULTICS and the 370/168, and LISP (under MACSYMA) on a PDP10 with a KL-10 processor. Although the arithmetic unit on the 168 has twice the speed of that on the KL-10, that group has chosen to do most of their work on the PDP-10 in LISP/MACSYMA, both because that factor of two is considerably diluted by the associated and inevitable non-numeric processing and because of the advantages of LISP over FORTRAN. (It should be noted that MACSYMA uses an algebraic notation, removing any notational advantage FORTRAN may possess over the standard LISP notation.) One complaint expressed to me by Charles Karney, another member of the group, was that LISP did not offer double precision reals or complex numbers. This is one area where FORTRAN's dedication to numerical applications puts it ahead of LISP. However, this shortcoming of LISP is in the category of implementation-dependent defects, and could be rectified without doing violence to the LISP language per se. Whether the department has the resources to rectify these defects is a question, however.

In none of the above arguments have we claimed that one cannot write LISP programs that are inefficient. In a language as versatile

as LISP it is inevitable that the user will want to take advantage of constructs that linguistically express perfectly what he is trying to say but computationally present obstacles to efficient code generation. Our position on this is that the default should be that the programmer feel no compunctions about using to the full the features of a language, but that on those occasions when it truly is the case that the machine's time is worth more than the programmer's (together with the time of those who have to read his code) then the programmer should know which constructions to avoid to permit the optimizer to do as good a job with his code as a good FORTRAN optimizer can do. Thus a systems programmer writing widely used systems code in LISP might as a general policy avoid heavy use of functions that do considerable work to keep the environment in a consistent state such as code associated with LISP's "special" variables. (If systems programming in LISP sounds like a contradiction in terms, it suffices to point to the MACLISP compiler, which is implemented in LISP. Much of the system code associated with Richard Greenblatt's LISP machine is also written in LISP.)

3. LISP uses a standard character set. Essentially all general-purpose terminals on the market now adhere pretty closely to the ASCII standard character set. It would be next to unthinkable for a language designer today to propose a language that made heavy use of a radically different character set, so this claim almost goes without saying.

4. LISP is interactive. If one wants to know the value of 1+1 while "talking to" LISP, one types (PLUS 1 1) (or 1+1) and LISP replies 2 without further ado. If one wants to get a big job underway, one simply invokes the top-level function of that job in

exactly the same way. And of course one's program can always type directly to the user and accept input from him at any time. Perhaps more significantly, one can interact with one's program while it is running, interrupting to both modify and/or examine the environment. In powerful languages like LISP, environment examination is made more complicated by the complexity of the environment; nevertheless LISP provides the tools needed to explore nested contexts and complex data structures.

5. LISP is modular. One of the joys of programming in LISP is that almost everything one does can be done incrementally, either on the user's command or under program control. If one is running a LISP program and wants to interrupt it to write another function, one can do so on the spot without having to re-read the whole program back into LISP. If one takes a dislike to the behavior of the lexical analyzer, its behavior can be modified on the spot, either locally or by wholesale and instantaneous replacement with a new analyzer. If the routine used by the top-level listen loop to print the answer is inappropriate to the task, it can be changed in one command; in fact, the entire top-level listen loop can be replaced. If a given system-defined function such as PLUS is not to the user's taste he can simply supply his own, without having to change every occurrence of PLUS in his program to a user-defined name. Even the READ function invoked by the top-level listen loop can be replaced with a user supplied function, an advantage so important that we afford it special treatment in the next section.

LISP's modularity is important not only to a single programmer but to groups of programmers cooperating on a project. When one develops a LISP program for a specific application, it can be used

later as a subroutine of somebody else's program. While this is true to a limited extent of most languages, it holds to a much greater extent in LISP. An example of this modularity concerns a program-proof-checker that Steve Litvintchouk, a graduate student of mine, has been writing in LISP. He complained to me that entering statements about programs into the computer was painfully slow because he was using standard LISP notation. So I made up a formal definition of the notation we had been using in class, implemented it one afternoon, loaded it into a LISP that already had Litvintchouk's program loaded (after a few debugging runs of course) and we were then able to talk to his program in the notation we wanted. (Sample: [Y:=X+5]<Y:=Y-1*>X=Y asserts that after setting Y to X+5, it is possible by iterating Y:=Y-1 to reach a state in which X=Y. We were tiring of writing (([] (:= Y (+ X 5))) ((<> (* (:= Y (- Y 1)))) (= X Y))) to express the same thing.) The remarkable thing about this particular exercise is that I had no idea what his code looked like at the time as I had not then gotten around to reading it, and he had no idea how one might go about changing notation in LISP. Yet despite this mutual ignorance, and without making <u>any</u> changes to his code, we were able to accomplish in a quite simple way what would require major surgery to the program in almost any other language.

6. LISP is notation-independent. Mathematically speaking, LISP programs form a set containing "atoms" and closed under the pairing function CONS. How such programs are to be represented is an implementation-dependent issue. In any implementation there are at least two representations, internal (consisting in the interpreted case of a graph whose nodes are computer words and whose edges are pointers, and in the compiled case of a string of machine instructions) and external (consisting traditionally of fully

parenthesized prefix (forward Polish notation) expressions). However, this does not exhaust the possible representations of LISP programs by any means, a point that is frequently over-looked and yet one that was made right at the outset by McCarthy, who used what he called MLISP notation, an algebraic notation that PL/I users would feel much more comfortable with than the fully parenthesized prefix notation. An implementation of MLISP exists at Stanford, and is the notation of choice for LISP users there. At MIT an MLISP-like notation called CGOL is available to the LISP user: at any time, even half-way through running his program, he can simply say (CGOL) and from then on he can rephrase

```
 (QUOTIENT (PLUS (MINUS B) (SQRT (DIFFERENCE (EXPT B 2)
                                              (TIMES 4 A C))))
           (TIMES 2 A))
```

as

```
(-b+sqrt(b**2-4*a*c))/(2*a)
```

or

```
(MAPCAR '(LAMBDA (I J) (PRINT (CAT '|Buy | I '| for | J '| dollars.|)))
        SHOPPINGLIST
        PRICELIST)
```

as

```
for i in shoppinglist, j in pricelist do
  print "Buy " ^ i ^ " for " ^ j ^ " dollars."
```

and so on. The versatility of LISP in comparison to most other programming languages becomes more apparent in an algebraic notation because a more direct comparison is possible without the distraction of having to allow for radically different styles of notation. (An aside having nothing to do with notation: although all programming languages deserving of the name can express the first of the above

examples, very few can cope with the second quite so directly.)

MACSYMA users also use an MLISP-like algebraic notation - in fact MACSYMA's parser is just the CGOL parser modified (by Michael Genesereth) to handle typed expressions. Unlike CGOL in plain LISP, MACSYMA notation is the default language for MACSYMA users.

The CGOL notation inherits LISP's modularity, in that it can be extended painlessly by the user even while in the middle of running a program. This legacy of LISP's puts this algebraic notation ahead of almost all other available algebraic programming languages with respect to syntactic extensibility. Only a few research systems come close to this level of convenience, such as Bell Laboratories' recently developed YACC (Yet Another Compiler-Compiler) system, a version of which has been developed by Alan Snyder at MIT where it is used as the "front-end" of Barbara Liskov's CLU language. Even these advanced systems do not offer the fast incremental extensibility of this syntactic front-end to LISP [6,8]. While this may at first appear to be due to some sort of breakthrough in extensible language work, it is really just a spin-off of LISP's excellent modularity.

It may be instructive to compare an ALGOL program taken verbatim from the Communications of the Association for Computing Machinery, Algorithm 482 [3], with its rendering in this algebraic dialect of LISP. We give the ALGOL version first, changing only its comment section for the sake of brevity.

comment We are given a set of towns numbered 1 to n. There are k one-way roads leading out of each town in such a way that if you ever go on a trip you can always get back home again, though not necessarily by

retracing your steps. However, it is not guaranteed that you can always get to the town of your choice. The problem is to group the towns into equivalence classes of mutually accessible towns.

The roads are represented by an array im[1:n, 1:k] such that im[r,q] is the q-th town accessible from town r. You are given two arrays ind[1:n] and orb[1:n] to store the results in. Town i is to be in the ind[i]-th equivalence class. Orb[i] is a list of towns arranged so that each equivalence class is in a contiguous block; the first town of each block is stored with its sign bit complemented (in particular town 1 will appear in orb[1] as -1) to distinguish the beginning of the block.

(The problem was stated in CACM in group-theoretic terms, with orb referring to orbits of group elements, but the ALGOL solution given in CACM solves the more general problem we have just described.);

```
procedure orbits(ind, orb, im, n, k);
  value n, k; integer n, k;
  integer array ind, orb, im;
begin
  integer q, r, s, j, nt, ns, norb;
  for j := 1 step 1 until n do ind[j] := 0;
  norb := 0; ns := 1;
  for r := 1 step 1 until n do if ind[r] = 0 then
  begin
    norb := norb + 1; ind[r] := norb;
    nt := ns; orb[ns] := -r; s := r;
a:
    ns := ns + 1;
```

```
    for j := 1 step 1 until k do
    begin
      q := im[s,j];
      if ind[q] = 0 then
      begin
        nt := nt + 1; orb[nt] := q; ind[q] := norb
      end
    end;
    if ns ≤ nt then
    begin s := orb[ns]; go to a end
  end
end
```

The following is the LISP rendering of the above procedure, using the algebraic dialect. For direct comparison we have adhered as closely as possible to the layout of the above program.

```
define "ORBITS"(ind, orb, im, n, k);
(
  new q, r, s, j, nt, ns, norb;
  for j in 1 to n do ind(j) := 0;
  norb := 0; ns := 1;
  for r in 1 to n do if ind(r) = 0 then
  (prog;                    % necessitated by the presence of (ugh) goto %
    norb := norb + 1; ind(r) := norb;
    nt := ns; orb(ns) := -r; s := r;
a;
    ns := ns + 1;
    for j in 1 to k do
    (
```

```
      q := im(s,j);
      if ind(q) = 0 then
      (
        nt := nt + 1; orb(nt) := q; ind(q) := norb
      )
    );
    if ns <= nt then
    ( s := orb(ns); go a )
  )
)
```

The only differences in this example are:

ALGOL	LISP	
a[b]	a(b)	(arrays)
begin end	( )	(block delimiters)
integer	new	(type declarations inessential)
procedure (and related text)	define (and related text)	
for i := 1 step 1 until n	for i in 1 to n	
a: (and associated goto)	a;	(and associated prog and go)
value	redundant (arrays may be values in LISP)	
≤	<=	(≤ not in ASCII standard)

None of these differences are particularly significant. However, we were fortunate that the goto did not leave a block, which would have precluded as direct a translation.

The attentive reader may have noticed that the "for i in" construct used earlier to scan a shopping list was used in this example to scan the list of integers 1 to n. If the user types the

expression "1 to n" by itself, he will get back that list; for obvious reasons the system does not do this explicitly in "for i in 1 to n" but instead just steadily increments the variable i to simulate scanning that list. This supplies a nice example of how one can unify language constructs (in this case the two notions of scanning a list and successively incrementing a register) if one is willing to let the optimizer (in this case, a source-level optimizer) take over the task of deciding whether scanning or incrementing is required. This is already an essential point in APL, where for some programs the naive implementation can be disastrous, for example where one wants to search the vector iota 1000000 (or 1 to 1000000 in LISP notation) for the number of integers less than a million expressible in two ways as the sum of two cubes. A smart APL interpreter will only generate and keep around as much of iota 1000000 as it needs at any one moment, avoiding demanding an unavailable amount of memory.

Let us emphasize again the role the notation plays in this example, which is to show that a LISP program need not be very different from a typical ALGOL program. To stress that the notation only supplies some window-dressing in this demonstration, we restate the above program in the notation preferred by the majority of LISP users.

```
(DEFUN ORBITS (IND ORB IM N K)
       (PROG (Q R S J NT NS NORB)
             (DO ((J 1. (ADD1 J)))
                 ((GREATERP J N))
                 (STORE (IND J) 0.))
             (SETQ NORB 0. NS 1.)
             (DO ((R 1. (ADD1 R)))
```

```
((GREATERP R N))
(COND ((ZEROP (IND R))
       (PROG NIL
             (SETQ NORB (ADD1 NORB))
             (STORE (IND R) NORB)
             (SETQ NT NS)
             (STORE (ORB NS) (MINUS R))
             (SETQ S R)
        A    (SETQ NS (ADD1 NS))
             (DO ((J 1. (ADD1 J)))
                 ((GREATERP J K))
                 (PROGN (SETQ Q (IM S J))
                        (COND ((EQUAL (IND Q) 0.)
                               (SETQ NT (ADD1 NT))
                               (STORE (ORB NT) Q)
                               (STORE (IND Q)
                                      NORB)))))
             (COND ((NOT (GREATERP NS NT))
                    (SETQ S (ORB NS))
                    (GO A)))))))))
```

The above comparison is a little like having a race between a Ford and a Porsche where the drivers are required to behave identically. After a few seconds the Porsche driver starts to grumble about being in third gear when he should be in fifth. In the above example there are some clumsy programming constructs that are the result of programming in a language that does not provide adequately for irregularly structured data. Actually, the input of this example (the array im) is regularly structured, but the result (the array orb) attempts clumsily (using the sign bits of its entries) to represent

.the irregularly structured answer.

We give below another version of the same algorithm, this time relaxing the constraint that we have to mimic the ALGOL solution as closely as possible. First we observe that the array im is really the only input data required. Second, we suggest that im be represented as a vector of lists, allowing a variable number of roads to leave each town. (In the group-theoretic special case of this problem, k is fixed, corresponding to having k generators of an n element group, but the solution given in CACM makes no essential use of k being fixed.) Third, we suggest that the answer be an array (corresponding to ind in the above program) whose i-th element is a list of towns accessible from town i (numbering these equivalence lists as in the above programs seems pointless).

All variables except adi (array dimensions of im) correspond to variables used in the above programs, though they may not be of the same type.

As before we give the algorithm in both notations, with the preferred notation first this time to avoid giving to much prominence to the CGOL notation, which plays no essential role in the point being made by this second LISP version of the program.

```
(DEFUN ORBITS (IM)
       (PROG (ADI IND N NT)
             (SETQ ADI (ARRAYDIMS IM))
             (SETQ IND (APPLY 'ARRAY (CONS NIL ADI))
                   N (SUB1 (CADR ADI)))
             (DO
```

```
        ((R 1. (ADD1 R)))
        ((GREATERP R N))
        (COND
         ((NULL (IND R))
          (STORE (IND R) (SETQ NT (LIST R)))
          (MAPC
           '(LAMBDA (S)
                    (MAPC
                     '(LAMBDA (Q) (COND ((NULL (IND Q))
                                         (STORE (IND Q) (IND R))
                                         (RPLACD NT (LIST Q))
                                         (SETQ NT (CDR NT)))))
                     (IM S)))
           (IND R)))))
       (RETURN IND)))
```

The CGOL version of the above, for the sake of those few who find something objectionable about the above notation, is:

```
define "ORBITS" im;
let adi = arraydims im;
let ind = array(nil : adi), n = cadr(adi) - 1, nt = nil;
for r in 1 to n do if null ind(r) then
  (ind(r) := nt := [r];
   for s in ind(r) do
      for q in im(s) do if null ind(q) then
        (ind(q) := ind(r); cdr nt := [q]; nt := cdr nt));
ind
```

The DO-loop (the outermost for-loop in the CGOL version) looks

for towns not yet in equivalence classes (LISP initializes untyped arrays to NIL, whence the NULL test). When a new town R is found, a length-one equivalence list (LIST R) (in CGOL: [r]) is started for it, and NT is set to the last LISP cell of the list (which initially is also the first). Then for each town S on the list, towns reachable from S that as yet belong to no list are put at the end of this list. (Since "(MAPCAR ... (IND R))" does not make a separate copy of the list (IND R) before scanning it, putting more towns on the end of the list forces the MAPCAR to consider those towns as well, which is what we want here. Though this is not good LISP style, it is how one would use LISP to do what was done in the ALGOL program, which is not good style either.) When all towns have been put into classes, the array IND is returned. To use the subroutine on input array X to find out what towns are accessible from town I, say ((ORBITS X) I) which will compute the IND array and then apply it to I. This of course is an inefficient use of ORBITS; usually one will say (SETQ X (ORBITS Y)) and later say (X I) for each I of interest.

For those readers wondering how hard I had to look to find an example better coded in LISP than ALGOL, I should say that I simply selected the most recent short ALGOL contribution to CACM's algorithms section that I could find on my bookshelf. I originally had not intended to give the second LISP version, but could not resist it once I saw what the algorithm was up to.

It must be admitted that any notational change to LISP raises the question of what constitutes a programming language. Must one buy lexicon, syntax, semantics and object code as a package, or can one shop around like a purchaser of a component stereo? The argument of this section has assumed that one <u>can</u> shop around, but while component

stereos may make sense to an electrical engineer, the concept of component languages may require some getting used to. This issue drives home the disadvantage of most languages, that you cannot use it for its good features without also having to accept its bad features. The situation has encouraged an attitude among language users that permits arguments of the form, "We can't have X because it has a bad feature, namely its syntax." As the computer-using community matures, it should grow more accustomed to the idea of purchasing language components and assembling them into their "ideal" system. As a spin-off from this sort of technology, we may hope to see a decrease in the number of languages available, a number which from this point of view we can attribute to the multiplicative way in which options must increase when you cannot order systems component by component. For example, if the community demands two kinds of lexicons, two kinds of syntax, two kinds of semantics and two kinds of code-generators, the market need supply only eight components in place of sixteen systems. Any increase in the sizes of the component options results in a rapid widening of this gap.

In this context the question of whether to choose a widely used language must be restated as whether to choose a widely used component. This question is perhaps most important for the syntax component, which for the user is as at least as visible as any other component. Although I suggested above that CGOL or MACSYMA notation offers the algebraic-notation user an alternative to the standard notation, other notations are equally possible, such as the very widely known ALGOL notation. In fact, an ALGOL front-end has been built for LISP (by Camilo Rueda) that adheres as closely to the Revised ALGOL 60 report as the ASCII character set will allow (omitting dynamic own as is customary). If the question of whether

the notation was widely known became a serious issue, the installation of ALGOL notation is straightforward, modulo tortuous implementation when the (rarely used in practice) environment-handling capability of ALGOL is given full throttle.

LISP's independence of choice of notation may be of value in an educational environment where, although a commitment to a single language may have already been made, individual instructors may nevertheless have a requirement for a different notation. LISP's ability to change notations in midstream without having to change languages reduces the overhead associated with maintaining several entire languages and their supporting maintenance teams and other paraphernalia. For example, if APL were needed on occasion, it would not be impossible to embed it in LISP; indeed a very compact definition of APL is possible in LISP. However it would seem only fair to ask the users of such components to assume their maintenance costs. There is also the question of whether it is fair to ask the students to learn one notation after another as they proceed from one course to the next.

7. LISP is applicative. That is, one can write non-trivial programs in LISP using only functional application. More significantly, this style can be used in LISP for programs that in most other languages would have to be written iteratively or recursively. The two LISP functions (strictly, functionals or combinators) that supply this power are MAPCAR and APPLY. MAPCAR permits a function to be applied to the elements of a list one at a time (coordinate-wise operation) while APPLY permits an operation such as PLUS to be applied to all the elements of the list (APL's notion of reduction, written +/a). APL, like LISP, offers non-applicative

features such as assignment and goto, but its users are strongly encouraged to rely on the applicative part of APL, and (presumably) to ensure that this happens APL offers a bare minimum of non-applicative control structures. A significant benefit of this style is that reasoning about such programs can be done in the same algebraic formalism that we have all been raised on since birth, instead of having to invent systems of logic especially to cope with the non-algebraic control structures of conventional programming languages. For example, knowing that + is associative even when applied to equal-length vectors as opposed to scalars, we can immediately see that (u+v)+w computes the same vector as u+(v+w), whereas we may have to argue more indirectly about the effect of the corresponding two programs in a non-applicative (in this case non-vector-manipulating) language. Vector spaces have been well studied and their properties carry over readily to reasoning about APL programs.

Let us give some examples where LISP can be used as an applicative language. Using (APPLY (FUNCTION PLUS) A) for APL's +/A, where the APL vector A is represented as a list in LISP, and using (MAPCAR (FUNCTION TIMES) A B) for APL's A*B, which multiplies vectors A and B coordinatewise to yield a new vector of the same length as A and B, we may get the effect of APL's +/A*B, which computes the inner product of two vectors, via (APPLY (FUNCTION PLUS) (MAPCAR (FUNCTION TIMES) A B)). (For algebraically inclined users, CGOL offers the variants a[b] for (MAPCAR (FUNCTION A) B) and a(b) for (APPLY (FUNCTION A) B). a[(b)] denotes the composite of these two, realized in LISP as (APPLY (FUNCTION MAPCAR) (CONS (FUNCTION A) B)).) One way to write a one-line matrix multiplication routine in LISP (yes, APL has no monopoly on one-liners) would be (in CGOL):

for i in x collect for j in list[[y]] collect plus[times[i,j]] .

which in the standard notation is

```
(MAPCAR '(LAMBDA (I) (MAPCAR '(LAMBDA (J) (APPLY 'PLUS (MAPCAR 'TIMES I J))))
                              (APPLY (FUNCTION MAPCAR) (CONS (FUNCTION LIST) Y)))
        X)
```

(The rather complicated thing being done to Y, which is a list of lists of numbers representing a list of rows of a matrix, is simply transposition.)

In some respects LISP's applicative ability is superior to APL's. APL lacks a specific operator that permits coordinate-wise application of a scalar operator, but rather relies on the fact that a scalar operation is being applied to a vector to deduce that coordinate-wise operation is called for. When a user has defined an operation that applies equally well to scalars and vectors, he cannot specify to APL whether or not he wants to apply his operation to the list as an entity or coordinatewise to its individual components. In LISP one distinguishes these cases explicitly as (A B) versus (MAPCAR (FUNCTION A) B) (in CGOL, a(b) versus a[b]). However, APL does distinguish reduction (e.g. +/a) explicitly.

8. LISP is widely used in academia. In a large portion of the academic computing community LISP is either a first or a second language. I received last week a letter from Gene Freuder, a recent MIT graduate now teaching at Indiana, where he is their AI representative. He was happy to report that in his department "everyone speaks LISP as a mother tongue." While one might not be

surprised to hear this about a department with a heavy AI bias, it is a tribute to LISP's ubiquity that it should be so honored in a department noted primarily for its work on programming languages and multiple-valued logic. LISP enjoys considerable use at Stanford University, Stanford Research Institute, Xerox Palo Alto Research Center, Carnegie-Mellon University, Bolt Beranek and Newman, IBM Yorktown Heights (for their SCRATCHPAD system), and can even be found as far afield as the University of Edinburgh and Japan's Electro-Technologica Laboratories. It is a sine qua non for any laboratory planning to embark on AI research, as it is the lingua franca of AI. Three of the above institutions (Xerox PARC, IBM and ETL) are only semi-academic, illustrating that LISP is not confined to universities alone.

9. LISP is used by half the MIT Computer Science faculty. Thus choosing LISP as the main departmental language, if one language is to be chosen ueber alles, would seem to involve the least upheaval. Further, LISP as a high quality language attracts high quality maintenance personnel. LISP has been maintained here not only by Jon L. White, a very experienced LISP systems programmer, but by some of the sharpest graduate students in the world. Guy L. Steele, winner of the 1975 George Forsythe student paper competition [10], the main student-paper competition in the computer-science world, has been a shining example of such help for several years dating back to his undergraduate days. The situation seems simply to be that the best languages attract the best students. Unless MIT suddenly experiences a dearth of good students, a fate few of us want even to contemplate, this high quality maintenance should continue at or near its present enviably high level.

10. No other language enjoys all the above advantages of LISP. This remains true even if advantages 2 and 4 are omitted. Let us argue this point on a language-by-language basis, choosing (at the risk of offending all whose languages have been omitted) FORTRAN, COBOL, ALGOL 60, PL/I, APL, ALGOL 68 and PASCAL. In the following we omit reference to items 2, 4, and 9; the first two are easy to satisfy while the last is obviously impossible.

FORTRAN: This fails on items 1, 5, 6 and 7. FORTRAN's storage allocation facilities and control primitives are too rudimentary to make FORTRAN useful outside the domain of numerical arrays, for which it was designed. By bending over backwards one can do anything in FORTRAN, as in any universal language (in Turing's sense); for example Joseph Weizenbaum implemented SLIP, a list processing language, in FORTRAN, and it is the language used for symbol manipulation at Bell Laboratories. However, one is sufficiently hemmed in by petty restrictions and inadequate data and control structures that no very strong case can be made for it.

COBOL. COBOL has something to offer academia that FORTRAN lacks, and that is a "data division" (to use COBOL terminology) that permits the user to define a rich variety of data types. This can lead to considerable simplification of the "program division," since the COBOL compiler can automatically make many decisions about where to put things and how to represent them. Unfortunately COBOL's data types are rich in about the same sense that an Eskimo finds a richness of snow varieties in the Arctic; this richness goes unappreciated in other climates, and COBOL's concern with business data processing makes it too narrow for use in other environments. (In this respect SIMULA 67 fares much better.) We can rule out COBOL on the same

grounds as FORTRAN, together with item 8.

ALGOL 60. In item 6 we observed a serious lack of versatility in ALGOL 60. This however was a problem having to do with ALGOL's rather limited data types and lack of first-class citizenship for arrays. In contrast ALGOL's control structures are remarkably powerful; one can do truly astonishing things with ALGOL's goto statement, such as jumping out of a procedure body that has called itself recursively to a great depth, resulting in exiting from all those levels of recursion in response to one goto. Also ALGOL offers call by name, which permits such useful constructs as Jensen's device for implementing a summation operator. However, not only does LISP offer facilities with all of this power, but it packages the facilities better. For example, exiting from several levels of recursion at once in LISP is requested explicitly with the THROW operation, which "throws" a value to a corresponding CATCH operation whose argument was responsible for invoking the THROW. Moreover these strengths of ALGOL, such as they are, do not make up for its weakness in the versatility of its data types. ALGOL fails on items 1, 3 (to a small but not terribly important extent), 5, 6, and 7.

PL/I. PL/I is nothing if not versatile. At least that is what IBM had in mind when they designed it. However, to be versatile without being modular is to be a super-market, where sometimes you spend more time looking for the aisle you need than all the other operations combined. PL/I fails on items 5, 6 and 7, as well as 1 in my opinion.

APL. APL passes strongly on item 7 (applicative), though not without a black mark for being unable to distinguish ordinary from coordinate-wise application explicitly. For what it attempts to be,

namely a vector-manipulating language, it also does well on item 1. Though we promised to omit reference to item 4 (interactive), this is a very strong feature of APL, and supplied me with my one reason to use APL considerably during a summer visit to IBM. Unfortunately, little of my work happened to fit the vector mold very well, despite the fact that most of it was heavily numerical, and I found myself from time to time using the embryonic 360 LISP system maintained by Fred Blair at IBM, on the ground that the additional time I had to spend running back and forth between the CP/CMS editor and LISP was made up for by the considerably decreased programming time in LISP. Thus I would fail APL on item 1, though not as seriously as the above languages. What really makes APL totally unacceptable is the insistence on a character set so out of touch with the ASCII standard that the department would be locked into an unacceptably inflexible (not to say expensive) situation if it were to generally adopt APL. APL also fails on 5 and 6.

ALGOL 68. This language is full of nice ideas. It has been carefully thought about by people with a concern for elegance and mathematical precision. Unfortunately the former has been sacrificed to the latter, and to learn ALGOL 68 requires considerable patience while one discovers the correspondence between one's programming intuition and the picturesque vocabulary of an ALGOL 68 programming guide (not to be confused with the language's formal definition, which is written in an almost inaccessible meta-language). ALGOL 68 fails on items 6, 7 and 8.

PASCAL. Nicklaus Wirth sensibly designed PASCAL to be simple in concept, easy to implement, efficient, yet versatile in its data types. Moreover, he saw to it that a good implementation on a machine

commonly found in universities (a large CDC machine) was made available. As a result, PASCAL has attracted the attention of many computer science departments as being an ideal pedagogical language. PASCAL's greatest drawback is the extent to which Wirth compromised (unnecessarily in my opinion) to achieve ease of implementation. As a result, PASCAL (like every other language above) lacks the first-class-citizen property that makes LISP so pleasant to use yet simple to implement in the event that you are willing to forego efficiency on some of the data types. I believe that this concession to efficiency, while it has many merits, detracts considerably from PASCAL's otherwise excellent versatility. As we have remarked elsewhere, efficiency should be a concern of the compiler as far as possible. Nevertheless, PASCAL remains among the more attractive possibilities.

B. DRAWBACKS OF LISP.

Most of the complaints of this section are implementation-specific and do not inhere in the LISP language itself. The one major exception to this is LISP's typelessness. It is unlikely that future implementations of LISP will clear up this issue without introducing substantial incompatibilities with existing LISP software. Moreover, many feel that typelessness is more virtue than vice. In contrast, the various complaints about the implementation cannot be taken seriously from a long-term stand-point since they are of a fairly trivial nature (except for the FUNARG problem). Moreover, other implementations of LISP are presently in an experimental stage, one in hardware (Richard Greenblatt's LISP machine) and one using the lexical scoping of the lambda calculus (Guy Steele and Gerald Sussman). Thus implementation-specific properties of LISP are at

present in a state of flux, and taking them into account in deciding that LISP was inadequate would be a case of throwing out the baby with the bathwater. Let us now begin with my one implementation-independent complaint.

LISP does not in any systematic way permit the user to specify the types of his data and variables. In fact some types are implemented directly as other types without LISP being able to tell which type is intended. For example, NIL serves double duty as the boolean FALSE and the empty list, as well as being recognized by LISP as an atom (though not one that can be used as a variable). And bit vectors are just single-precision integers; indeed only the presence of bit-manipulating operations permits LISP to claim that it has the type bit vector. Though this typelessness of LISP has an advantage in permitting the user to save programming time by not having to declare types, there remains the fact that many bugs are detectable because they violate type constraints. These violations will be caught by the interpreter (if at all) only when the offending portion of the code is actually run; for this reason some LISP users compile their newly written code before or even instead of debugging it interpretively as a first debugging step on the ground that at least the compiler does a fair to moderate job of detecting type violations. This philosophy of tight control over types is at the heart of the design philosophy of Barbara Liskov's CLU language at MIT along with similar research languages at other campuses such as Carnegie-Mellon's ALPHARD language.

Many LISP users do not agree on this point, and feel that types, like gauges, cause more trouble than they cure by being themselves the major bugs in the users program. Also, for beginning

programmers, and for people wanting the minimum "hassle" while trying to write a program that they know will be correct on account of its extreme simplicity, the added burden of having to declare types is unreasonable. Thus this objection is at present controversial.

Another source of troubles with LISP is the classic FUNARG problem, so named by Joel Moses [4] in an early discussion of the problem. Although a completely correct handling of this problem is not at the top of all users' lists of demands, it is for some; also, the formal description of LISP is simplified if one assumes that the problem, which involves being able to pass around program-environment pairs just like any other data, is properly taken care of. As things stand at present, the partial solution implemented in MACLISP is better described operationally, leading to a more clumsy description of LISP than is possible otherwise. Several plausible solutions (by Richard Greenblatt, Henry Baker, Guy Steele and Gerald Sussman) are in the air, and one may soon find its way into MACLISP.

A much less serious complaint is that the user may not specify a lower bound for any array dimension other than 0. To an extent this objection can be overcome using the notational flexibility discussed in section 6 by permitting a(i) to denote (A (PLUS I 7)) or whatever. Another complaint is that with the bit-vector data type, LISP itself only offers 36-bit bit vectors. However, LIBLSP (the public LISP software library on the ITS machines) offers a package written by Henry Baker that confers efficient unlimited-length bit-vector processing capability on LISP. This package really ought to be available directly to LISP users. In my LINGOL natural language system, which does a considerable amount of set-oriented processing involving set intersection and union, bit vectors have supplied a very

efficient implementation of sets.

A continual source of petty irritation for me is the compiler, which overlooks many simple optimizations. However, my pique notwithstanding, the compiler is probably the best LISP compiler available anywhere as it does a remarkably good job of optimization considering LISP's versatility. No matter what language the department settles on, if it is as powerful as LISP, people will be complaining about overlooked "obvious" optimizations for quite a while, possibly for ever.

One can come up with a variety of minor complaints of this ilk, but beyond the first two major complaints about LISP's typelessness (regarded by many others as a virtue) and lack of environment-manipulating capability, I personally am pretty satisfied with the language.

C. DEPARTMENTAL REQUIREMENTS.

Though the department has asked for a good educational language, there is no harm (if we are considering LISP) in asking that the research needs of the department also be considered. Here are a number of headings (suggested to me by Ronald Rivest) under which one may ask about the utility of LISP.

1. Desktop calculators. As pointed out before, one need merely type (PLUS 1 (TIMES 2 3)) (though this is a case where one of the algebraic notations, permitting 1+2*3, would seem preferable) to get the answer 7. Although my secretary cannot program she has used LISP regularly for two years for precisely this application, even though the bulk

of her arithmetic is just adding up numbers. In fact almost any expression that can be typed on an SR-51, say, can be typed almost with no change to LISP using CGOL. Moreover, the comparatively unlimited storage of LISP is available for storing intermediate results. Thus a LISP terminal consisting of a calculator-sized keyboard and a 15-digit display would already provide enormous power. Going to a 30-character alphanumeric display would increase the utility of such a terminal considerably. Such terminals could be so cheap that each office in the building could have two or more if necessary. With reasonable system design, the formalities of logging such terminals on and off could be dispensed with. Thus LISP would be available as a powerful replacement for programmable calculators, yet lacking none of their convenience. The possibility exists of making such terminals completely portable, at least within the building, relying on a radio link for contact with the department's system.

2. Number-crunching. We have already referred to the department's Plasma Dynamics group, in the section on efficiency. This supplies an example of where LISP is already in use within the department for "number-crunching" on a large scale.

3. Classroom language. Not surprisingly, LISP is the classroom language for Patrick Winston's AI course. It is also one of the languages taught in 6.031. These are two of the department's core course. In addition LISP is used (admittedly with the CGOL dialect) in 6.046, my algorithms course; using any other widely available language I would find many of the algorithms I teach awkward to express.

4. Publication Language. This is one area where the case for LISP is

weak. However, the committee has already conceded that a language need not be rejected as a teaching language just because there exists a large body of software written in some other language. In other words, on the software-input side, language is not a serious consideration from the committee's point of view, and they recommend that FORTRAN be made available for software-input without at the same time inflicting it on the students. It would seem reasonable to apply the argument to software-output as well, making available a publication language (say FORTRAN, to return tit for tat, though seriously PL/I is probably a better choice) for the purposes of debugging programs about to be published.

In summary, I would say very simply that LISP would make an excellent departmental language. All things considered, it has little serious competition from any language except PASCAL, and even that competition is minimal. At this point it is appropriate to include the ad hominem argument that LISP, an MIT product, has had a considerable impact on the academic computing community over the past decade and a half, and along with magnetic core storage, CTSS, MULTICS and MACSYMA has been responsible for making MIT among the world's most influential sources of Computer Science ideas.

Bibliography.

[1] Fateman, Richard J. "Reply to an Editorial." SIGSAM Bulletin 25, 9-11. (March 1973).

[2] Hewitt, C. E., P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence," Proc. IJCAI 3, p. 235. 1973.

[3] McKay, John and E. Regener. "Transitivity Sets." Algorithm 482, CACM 17, 8, 470. (August 1974).

[4] Moses, Joel. "The Function of FUNCTION in LISP." AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[5] Popplestone, R. J. "The Design Philosophy of POP-2." Machine Intelligence 3 (ed. D. Michie), 393-402, Edinburgh U. Press, 1968.

[6] Pratt, V. R. "Top Down Operator Precedence." Proc. ACM SIGACT/SIGPLAN Conf. on Principles of Programming Languages (POPL 1), Boston. (October 1973).

[7] --------. "A Linguistics Oriented Programming Language." Proc. 3rd International Joint Conference on AI, Stanford, 1973.

[8] --------. "CGOL - an Alternative External Representation For LISP Users." MIT AI Lab Working Paper 89. 1976.

[9] Randell, B. and L. J. Russell. ALGOL 60 Implementation. Academic Press, London, 1964.

[10] Steele, Guy Lewis Jr. "Multiprocessing Compactifying Garbage Collection." Comm. ACM 18, 9, 495-508. (September 1975).