D. Programming

R67-22 The LISP 2 Programming Language and System—P. W. Abrahams and C. Weissman (*1966 Fall Joint Computer Conference, AFIPS Proc.*, vol. 29, pp. 661–676).

LISP 2 will one day stand on its own legs and be evaluated on its own merits. But in its first public appearance it is in the role of a child of famous parents. Comparisons are invited when a child is brought into the world to carry on and eventually take over his parents' business. Of course, people get old and eventually retire, if for no reason other than that they can no longer do what they were once able to do. Computer languages do not degenerate in the same way. When they are deliberately replaced it must be that the new generation addresses itself to problems that are *effectively* outside the scope of the old one. This point is taken up again below.

LISP holds an honored place alongside FORTRAN, ALGOL, and IPL-V as an important tool and, what is more important in the long run, as a seminal construct. That the culture spawned by FORTRAN and ALGOL is ubiquitous is obvious and need not be further discussed here. The number of people fluent in IPL-V is very small. The importance of that language in historical context is that it explicitly dealt with and exploited dynamically changing data and program *structures*. It planted the list-processing seed. At least traces, and usually much more, of its issue can be found in every modern product of the computer world. The LISP culture also has few active participants. However, it is immensely vigorous. One sign of this vigor is that new LISP interpreters and compilers appear with considerable regularity. Another is that extremely difficult problems (e.g., symbolic integration, visual object recognition, programmed proof procedures, mathematical assistants) continue to be cast in LISP programs. But the spermatic character of the LISP idea reveals itself in that it has permeated research in the foundations of computation. LISP *is* a topic in recursive foundation theory. It is *built on* the lambda calculus. Mathematical theorems *about* and *in* LISP are not ad hoc or a posteriori constructs clothing things not basically logically structured in mathematical costume.

What is the beauty and from whence derives the power of LISP? The beauty of LISP is that it is so utterly *simple*. The first few pages of the LISP 1.5 manual tell all that really needs to be told. All the rest is elaboration and example. The word for that is "elegance." The power of LISP is a function of its elegance. LISP is *not* a list processor. To be sure, its basic internal data type is the list structure. Indeed, the programmer may manipulate list structures explicitly. But large programs are written in which explicit list manipulation is not the major issue. The essence of LISP is that it is a *functional* language. And by this is meant that it applies functions to arguments. At this stage of the development of computation it almost need not be said that the latter may again be functions applied to arguments. It is, however, altogether uncommon that functions may themselves be applicative expressions, that, in other words, a program may execute a process (evaluate an expression) the result of which is a function (not a function name) which may then (or later) be applied to arguments. That unlimited nesting of such expressions within expressions is permitted goes without saying. Similarly for the fact that functions may be recursive in complicated ways. However, another word should be said about recursion. This is that the basic list organization of LISP and the provision of the operators CAR and CDR which yield the first element of a list and a list with its first element removed, respectively, makes recursive operations on complex data structures quite easy and, more importantly, natural.

Another strength of LISP must be mentioned because of what follows: It is possible—even normal—to have a program *construct* an expression and then have that expression evaluated. In other words, data can be treated as program. It is, of course, also true that a LISP program may operate on itself, i.e., that program may be treated as data.

Granted all the beauty, elegance and power described here, what then motivated anyone to create a child to replace this famous person? That LISP 2 is so intended follows from its name. Does the new package repair certain faults of the old one or does it take advantage of new opportunities, perhaps new ideas, to enhance and expand it?

The authors criticize LISP 1.5 (the current form of LISP) for not having "a convenient input language" and for treating purely arithmetic operations inefficiently. The first of these criticisms is met by providing a preprocessor to convert very ALGOL-like source level programs to very LISP 1.5-like internal language programs. The internal language is an intermediate stage. It is finally compiled into machine code. The arithmetic inefficiency difficulty is met by introducing type declarations. These give the compiler sufficient information *about* the program being compiled to permit generation of efficient code.

Here, by the way, is the first public admission on the part of what has sometimes been called the LISP "priesthood" that there is anything "inconvenient" about LISP notation. Indeed, the heavily parenthesised LISP notation is sometimes seen as a blessing (even if in disguise) in that it keeps the priesthood small—the price of admission, i.e., having to learn to balance endless series of parentheses, being too high for most ordinary humans. The removal (or at least reduction) of artificial tariffs can be greeted only with pleasure.

The efficiency issue is more pervasive than appears at first glance. Behind its acknowledgement lies an even more important recognition—namely, that many important problems demand information processing facilities of disparate kinds, i.e., that "symbol manipulation" is not the only tool required by the serious workman. It is here that the word "effective" as used above becomes operative. For no one argues that LISP (or, for that matter, FORTRAN) is not "universal" in that programs prescribing the computation of any computable number cannot be written in it. But there is a difference between *practical* computability and computability. (This difference is, by the way, becoming every more important as the maximum delays people and things can be made to endure are more and more determined by real-time criteria.)

It is in the service of the newly valued catholicity that the major departures from classical LISP have been ordained. The starting point is that efficient arithmetic is desired. Type declarations permit compilation time identifier discrimination and thus aid in generation of code appropriate to each task, hence efficient code. But what kind of arithmetic? Well, integer, floating-point, Boolean, multiple precision, ... But the storage of multiple precision numbers raises difficult problems in dynamic storage management, precisely the same ones arising out of attempts to allocate and recover array and matrix space.

The problem is that programs will demand blocks of *contiguously addressable* storage at essentially unpredictable times. If fewer cells are demanded than are on the free storage list, but no *contiguous* block of the required size is free, then the program has run out of space. But, except for poor organization, space is actually available. That situation cannot be allowed to persist. The solution adopted by LISP 2 is a so-called "compacting garbage collector." As in classical LISP, abandoned data structures are permitted to accumulate as long as there is no shortage of free space. They are not collected until a new pool of available space *must* be formed. In the classical LISP garbage-collection cycle, abandoned cells are marked as being again available by being strung into one monolithic free storage list. The process is roughly analogous to stringing a thread through all newly available cells. In a sense, the thread moves but the cells stay in place. In compacting garbage collection, on the other hand, the number of cells so freed is counted and the data contained in that number of cells starting from the top of the original list of available space are moved into the freed cells. All appropriate pointers in all data structures are modified accordingly. The new free storage list then consists of the largest possible block of contiguously addressable cells. Clearly, this latter scheme may involve the transfer of considerable amount of information, hence, a proportionate expenditure of processing time.

One important advantage of dynamic storage allocation is that space is used over and over again, thus lowering the minimum storage requirements of a given program over what it would have had to have been under a rigid, static storage preassignment scheme, e.g., one governed entirely by DIMENSION statements. Nevertheless, there is some such minimum associated with any given program and the data structures on which it is to work. It is precisely in the complex problem areas for which LISP was designed that reliable estimates of such minima are very hard to make. The original list of available space is, therefore, made as large as possible—in batch processing usually the remainder of core after all essentials have been loaded. But the larger that list, the longer the garbage-collection cycle.

Recall that garbage collection is not initiated until the free storage list cannot satisfy a demand for free space—whether that demand be for a single cell or for a block of cells. But it is in the nature of the scheme here outlined that all substantive computation must cease while the list of available space is reorganized. The time required to collect garbage on a typical 7094 LISP 1.5 system is on the order of 0.5 second. That is for a machine with a 32 000-word memory of which about 10 000 words are devoted to available space. LISP 2 is meant for machines with possibly 256 000 words of core storage of which 200 000 words might well be allocated to the free list. Compacting probably doubles garbage-collection time. Thus, a garbage-collection cycle in a LISP 2 system running on a 256 000 machine of 7094 speed might take 20 seconds! Such lengthy interruptions of substantive computation will prove to be intolerable for precisely those problems which are now beginning to dominate the interest of the most advanced programming community, hence of the most natural LISP 2 customers. For those problems involve either intimate man-machine interaction (e.g., MATHLAB) or control of elaborate machines (e.g., ROBOT). They are, in other words, problems in real-time computation which may very well founder on interrupts of the indicated magnitude.

A better solution to the garbage-collection problem would have been to break the overall effort of space administration into small quanta and distribute these more or less uniformly over the whole program running time. The main objection to certain existing algorithms of this kind is that they demand explicit, i.e., programmed, erasure of data structures and thus place an unwanted burden on the shoulders of the programmer. But this can be avoided in block-structured systems. In any case, a new way has to be found.

Experience indicates that the solution to the problem of dynamic space administration in a list-processing system so pervades all other systemic issues as to virtually dictate overall system strategy. If this insight has any validity at all, then it would argue that the LISP 2 team should have started with a set of *functional specifications* derived from both the acknowledged power of the LISP concept *and* the environment in which the new system has to operate. Because that environment must of necessity include the facts of real-time application, time-sharing, machines with very large first level memories, and paging, they would have been led naturally to research on distributive compacting garbage collectors. Under such a regimen no implementation would have been started until that major problem was solved. Judging by the superb quality of the individual team members, it is hard to believe they would not have succeeded.

That LISP 2 permits a variety of data types and even the introduction of new data types can no longer serve to distinguish it from competing languages. AED and PL-I, to name only two, have similar facilities. And both permit unlimited recursion and partial word operations. And it has been shown that pattern-driven data-manipulation functions may be introduced into a large class of languages.

LISP's remaining claim to distinction is that "it is unique among programming languages in the ease with which programs can be treated as data." Of course, many languages can "treat programs as data." Even lowly FORTRAN can operate on FORTRAN programs! What is meant here is that a currently *regnant* program can be operated on *and the resulting structure again treated* as a program, i.e., evaluated—all within a single run. In other words, *that data can be treated as program!* (By an unconscionable stretch of the imagination one can conceive of FORTRAN achieving this mode of operation by means of overlays and chaining. That would hardly qualify as effective computing.)

This important feature has been kept in LISP 2, at a price, however, that goes some way toward defeating the "inconvenient language" component of the initial motivating argument. For programs that are to be subjected to such treatment must be in the intermediate, i.e., LISP 1.5-like language. The programmer who wishes to take advantage of this aspect of the power of LISP 2 must, therefore, be conversant in the intermediate language as well as in the ALGOL-like language.

It was said earlier that LISP 1.5 permits constructs of the following type:

$$((fa)b)$$

where f is a function to be applied to the argument a. The results of this application are again a function (as opposed to a function name) which is then applied to the argument b. Thus, for example, if

$$f = (\text{LAMBDA}(G)(\text{LAMBDA}(X)(G(GX))))$$

and

$$a = \text{SQRT}$$

then the result of applying f to a is the function

$$(\text{LAMBDA}(X)(\text{SQRT}(\text{SQRT}\ X)))$$

which if then applied to $b = 16$ will yield the value 2. Had f been applied to the LOG function, it would have yielded the LOG LOG function. This is an enormously powerful feature of LISP 1.5. Unfortunately, the paper under review fails to allude to its presence in LISP 2. Expressions of the kind just illustrated can certainly be written and are undoubtedly legal in the LISP 2 intermediate language. There is a significant question, however, over the possible implementation of such a mechanism within the LISP 2 framework. In LISP 1.5 the interpreter is always kept in core (even when dealing with compiled LISP 1.5 programs) to deal with just such issues. As there is no LISP 2 interpreter, the system must call the compiler whenever such an issue arises during the running of a program. This carries with it the two disadvantages that 1) considerable memory space must be left for the compiler (unless overlay techniques are used!) *and* 2) that non-negligible computing time is once again devoted to purely administrative matters. The overall effect of these penalties is to encroach on the freedom with which the LISP 2 programmer will use this powerful device. And the worst consequence of that is that it increases the burden of awareness which the programmer must carry. It is arguable that, at least for the problem classes relevant to this discussion, the ultimate limit of what can be programmed is determined by the weight of just this burden.

All the above views taken together point to the conclusion that the child has overcompensated for the reputed deficiencies of the father. ALGOL *is* a more convenient language than LISP 1.5 and disparate data types *are* highly desirable. But their inclusion has been achieved at the cost of weakening some of the muscles for which LISP is most deservedly famous.

The very caliber of the LISP 2 architects and builders give the disparity between the hope and the achievement a paradoxical character. How is it to be explained? It was stated earlier that the effort should have been grounded on a set of functional specifications and tuned to the emergent computing environment. What fault there is, is probably attributable to the atmosphere of *development* (even production) in which the task was executed. Under that kind of pressure, the truly elegant foundation of the LISP idea eroded. Then too, it should be appreciated that in comparison with LISP, ALGOL, however practical it may be, is a jungle of ad hoc rules and devices. No fine-spun creation could hope to preserve its charm in such surroundings.

But the wake over lost innocence must not be overly prolonged. The technical achievement of the LISP 2 group deserves the most keen respect and admiration. To say the system rests on nonparsimonious foundations is not to say that it is a patchwork. It is an integrated machine of tremendous versatility. If it lives up to its specifications, it should easily dominate such other eclectic systems as FORMULA ALGOL, FORMAC, and the various SLIP embeddings. Because of the huge advance sale of tickets to the PL-I extravaganza, there can be no contest between it and LISP 2. Whatever LISP 2's performance in popularity polls might eventually be, its ultimate significance must be judged in terms of its influence. This is the measure that established the fame and honor of its direct antecedants. The problems alluded to here wait to be solved. Perhaps the very people who created this system—and who once more demonstrated that a platoon of superior craftsmen is enormously more powerful than a brigade of plodders—may now meet the challenge they have so well illuminated.

JOSEPH WEIZENBAUM
Mass. Inst. Tech.
Cambridge, Mass.