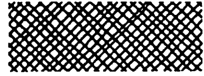
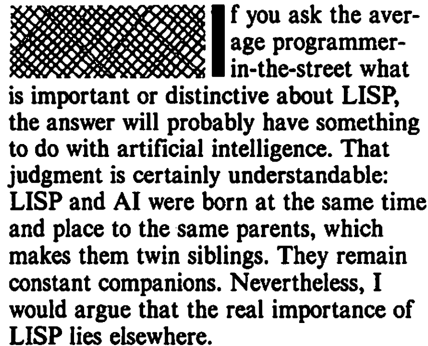


Mutant languages from the LISP lab

By Brian Hayes

APL is like a diamond. It has a beautiful crystal structure; all of its parts are related in a uniform and elegant way. But if you try to extend this structure in any way—even by adding another diamond—you get an ugly kludge. LISP, on the other hand, is like a ball of mud. You can add any amount of mud to it and it still looks like a ball of mud.

—Joel Moses, as paraphrased by Guy Lewis Steele Jr. and Gerald Jay Sussman[1]



If you ask the average programmer-in-the-street what is important or distinctive about LISP, the answer will probably have something to do with artificial intelligence. That judgment is certainly understandable: LISP and AI were born at the same time and place to the same parents, which makes them twin siblings. They remain constant companions. Nevertheless, I would argue that the real importance of LISP lies elsewhere.

What sets LISP apart, in my view, is its role as a language laboratory. LISP is a culture medium for computer languages: a nutrient broth where new ideas emerge from the ooze, then mutate, evolve, and cross-breed. Here I will review a few of the more interesting experiments to come out of the LISP laboratory.

Why LISP

Why has LISP been the focus of all this linguistic experimentation? A number of hypotheses have been offered:

Ivory-tower theory. Until recently, the economic significance of LISP programming was nil. There was little commercial pressure to standardize the language. Thus LISP evolved freely while other languages of the same generation, such as FORTRAN and COBOL, were frozen in an early stage of development.

Mother-of-invention theory. LISP has been used to attack hard problems, which demand powerful tools. LISP itself has not been adequate so there has been no choice but to build new and better languages.

Hotshot theory. LISP programmers are a bunch of malcontents who will not leave well enough alone. They all think they know better than anyone else how the language ought to work.

Rolling-mudball-gathers-lots-of-moss theory. LISP is so easily extended that it sprouts new languages willy-nilly. Every LISP program is a new version of LISP.

There may be truth in all of these propositions, but more substantial factors are also at work. One important point is that LISP has a simple syntax. Most of the work of computation in a LISP program is done by a single kind of expression, the function call, which takes the following form: *(function-name argument$_1$ argument$_2$ argument$_3$...)*. There is not much more to know about the syntax of LISP programs.

Other computer languages, in contrast, have given rise to strident debates over syntactic issues. (Is it better to bracket statements with *begin...end* or with {...}? Should the semicolon be a terminator or a separator of statements? What is the best delimiter for comments?) In the LISP community, questions like these have largely been ignored. Most of the languages that have grown out of LISP have adopted the syntax of the parent language with little change.

Dismissing the problem of syntax in this offhand way has had two important effects. First, it has focused attention on semantics—on the meaning of expressions rather than their form. Second, it has made the implementation of new languages within LISP much easier. Because parenthesized lists serve as a notation for both programs and data, the same expression can be interpreted as data by the underlying LISP system and as a program by the embedded language.

Parsing a program–which is a major undertaking for a language such as Ada or PL/I–becomes almost trivial. Most of the work is done by the built-in function *read*, which digests entire expressions in a single gulp. There is a certain irony in the longevity and stability of LISP syntax. The parenthesized prefix notation was introduced in 1958 only as a temporary measure to get the first LISP interpreter running.[2] The plan was to replace it as soon as something better could be devised. Evidently, nothing better has turned up yet.

Planning and conniving

Some of the most influential variations on LISP were conceived in the late 1960s and early 1970s by Carl Hewitt of the Massachusetts Institute of Technology. The first of his languages, called PLANNER,[3] added to LISP a facility for goal-directed programming.

A PLANNER program had two parts: a data base of assertions and a set of procedures for proving theorems about the assertions. For example, an assertion might state that Socrates is human, and a theorem might hold that anyone who is human is mortal. Given the goal of proving that Socrates was mortal, PLANNER would attempt to apply every theorem it knew to every assertion in the data base until it either satisfied the goal or exhausted the possibilities.

The idea of computing by proving theorems has now become familiar through the language PROLOG. PLANNER definitely influenced the development of PROLOG,[4] but the connection should not be overemphasized; there are fundamental differences between the two languages. In the first place, PLANNER theorems were expressed in procedural form, whereas PROLOG is a purely declarative language. Furthermore, PROLOG uses a more sophisticated theorem-proving strategy, called unification. (Soon after PROLOG appeared, of course, it was implemented in LISP.[5])

The most conspicuous innovation in PLANNER was its control structure: from the programmer's point of view, there was none. Procedures were invoked automatically by a pattern-matching process.[6] In proving the mortality of Socrates, any procedures matching the patterns *(mortal ?x)* or *(human ?x)* would be triggered. The sequence of procedure calls was not under the programmer's control. PLANNER always made a depth-first search of the data base and backtracked when it came to a dead end.

The full PLANNER language was never implemented, but a subset called MicroPLANNER was written in MacLISP (the MIT dialect of LISP) by Gerald Jay Sussman, Terry Winograd, and

Eugene Charniak.[7] Its most celebrated use was in Winograd's SHRDLU program, which answered natural-language queries about a world of toy blocks.

Eliminating all control structures took a lot of the bother out of programming, but a search based on blind backtracking did not make for dazzling efficiency. CONNIVER,[8] developed by Sussman and Drew V. McDermott, was a successor to PLANNER with provisions for explicit control of execution. CONNIVER was PLANNER with a manual transmission. Using the control facilities provided by CONNIVER, the system could be instructed to examine all assertions about Socrates before looking at those having to do with humans and mortals.

Acting

Hewitt's second linguistic experiment is called the Actors model of computation.[9,10] Actors are independent entities that encapsulate both data and procedures. A program (or a system of programs–the distinction is blurred) constitutes a society of Actors that communicate with one another by sending messages. Each Actor is defined by two components: the set of messages it recognizes and the set of acquaintances with whom it can communicate.

Just as PLANNER is reminiscent of PROLOG, the concepts underlying the Actors model will be familiar to those who know object-oriented languages such as Smalltalk. Actors correspond to objects in Smalltalk; actions are methods; messages have the same name in both languages. Once again, however, similarity is no guide to heritage. Actors and Smalltalk were invented independently and at about the same time; they both owe much to Simula, which was developed a decade earlier.

Hewitt and his colleagues implemented the Actors model in a language called PLASMA (for PLANNER-like System Modeled on Actors). PLASMA is built within LISP, although, as will be discussed later, it differs significantly from most versions of LISP prevalent at the time. Studies of the Actors model have continued, including a project called the Apiary,[11] which proposes a parallel computer architecture in which each Actor has its own processor. In recent years, others in the LISP community have adopted object-oriented programming with enthusiasm, and several dialects of LISP now have object-oriented extensions or sublanguages. The best known of these are LOOPS[12] (LISP Object-Oriented Programming System), Flavors[13], and the public domain XLISP.[14] (To add to the confusion, the recently introduced language ACTOR is object-oriented but has little to do with Hewitt's work in LISP; ACTOR is implemented in a Forth-like language.)

Scheming

One small feature of the PLASMA language, seemingly nothing more than a technical nicety, has had far-reaching consequences. To make the Actors model work properly, PLASMA was based on lexical scope rules, in which the value of a variable is determined by the textual context in which it is defined. Traditionally, LISP has employed dynamic scope rules, in which values are determined at the point where a variable is referenced. Lexical scope was essential in PLASMA because the acquaintances of an Actor are listed in the Actor definition and must not be altered by any rebinding of the same names elsewhere in the program.

In 1975 Sussman and Guy Lewis Steele Jr., while studying the Actors model (out of "morbid curiosity," as they put it), made a surprising discovery.[15] They had written a small Actors interpreter with lexical scope rules and had found that the constructs representing Actors took a familiar form: they were simply lambda expressions.

Lambda is the mechanism of procedural abstraction in LISP, the means by which an expression or a series of expressions is wrapped up to form a unit of executable code. For example, whereas *(+ x x)* evaluates immediately to twice the value of *x*, *(lambda (x) (+ x x))* yields a procedure that doubles the value it is given as an argument. Like an Actor, a lambda expression has two parts: the code to be executed—*(+ x x)* in this case—and the environment in which that code is to be evaluated. The code corresponds to the actions taken by an Actor in response to messages. The environment, which is a list of the variable bindings in effect when the *lambda* form is defined, corresponds to an Actor's list of acquaintances.

Out of this small observation Sussman and Steele (with later contributions by many others) constructed an entire new dialect of LISP called Scheme.[16] Among the experiments discussed here, Scheme is the only one yet to have escaped from the laboratory and proved its mettle in the wild. At last count some nine implementations were available, including some for microcomputers.

Like PLASMA, Scheme has lexical scope rules, but its most remarkable departure from established custom is the first-class status it accords to procedures. A first-class object in a programming language is one that has no arbitrary restrictions on where it can go and what it can do. A first-class object can be assigned as the value of a variable, passed as an argument to a procedure, returned as the result of a function, or stored in a compound data structure. In most languages only simple values, such as numbers, have all of these rights and privileges, and in earlier LISPs procedures were unquestionably second-class citizens. Scheme emancipates them.

The combination of lexical scope and first-class procedures encourages a distinctive style of programming in Scheme, emphasizing modularity and data abstraction. For example, a procedure named ***make-db*** might set up a private data base and then return another procedure as its value. When the latter procedure is called it has exclusive access to the data base. Moreover, each invocation of ***make-db*** creates a new instance of the data base and a new access procedure.

Naming

Just as Scheme liberates procedures from their second-class status, a language called Symmetric LISP is built on the concept of first-class environments. Symmetric LISP is currently being developed by David Gelernter of Yale University, Suresh Jagannathan of MIT, and Thomas London of AT&T Bell Laboratories.[17] An environment is essentially a dictionary in which one can look up any variable name and find the corresponding value or definition. A common way of implementing an environment is by using the data structure known in LISP as an association list, or a-list. The environment *((x 3)(y 4))*, represented here as an a-list, records two facts: that the variable *x* is bound to the value 3 and that *y* is bound to 4. If the procedure represented by *(lambda () (+ x y))* is executed in this environment it returns the value 7.

Gelernter and his colleagues point out that associating names with values is a central activity in many areas of computing. Structures analogous to environments are needed to support the global and local variables of any language with lexical scope. Packages or modules for separate compilation also establish namespaces. Records with named fields, as in Pascal, are merely a variation on the same theme. Even a directory of disk files is organized as a dictionary associating file names with file contents. In Symmetric LISP, all of these forms of naming (and others) are handled by a single data structure—the first-class environment.

The programmer's basic activity in Symmetric LISP is to build up nested environments in which names are bound to the values of arbitrarily complex expressions. Even lambda expressions are represented as environments, where the bindings are those of the formal parameters. Evaluating an environment yields a new environment whose bindings have been updated. The ultimate result of a

computation is an environment in which variables are bound to answers.

A major motivation for the development of Symmetric LISP is parallel processing. As a rule, all the name-value pairs in an environment can be evaluated concurrently. Where one binding depends on a value calculated in another binding, the conflict can be detected automatically; no special control structures are needed to synchronize parallel execution. Implementation of Symmetric LISP on a multiprocessor system is currently under way; at the moment a Symmetric LISP interpreter runs (without parallelism) on a sequential LISP system.

Xapping

Parallel processing has become a major preoccupation of the LISP community in recent years, and Symmetric LISP is by no means the only result. Several other language proposals take the control-structure approach to parallelism. An example is Multilisp, created by Robert H. Halstead Jr. of MIT.[18]

Multilisp is a dialect of Scheme augmented with constructs for the control of parallel execution. The most important of these constructs is called *future*. An expression of the form *(future x)* has two effects: it starts the calculation of *x*, which can be any expression, and it also immediately returns a placeholder value, which will eventually be replaced by the actual value of *x*. Hence, *(cons (future x) (future y))* would launch the calculation of both *x* and *y* while simultaneously allowing *cons* to return a placeholder value to its caller.

Still another form of parallelism is being explored by Steele and W. Daniel Hillis of Thinking Machines Corp.[19] In Connection Machine LISP they introduce a new data structure called a xapping that serves to organize fine-grained parallelism (the concurrent execution of many small operations rather than a few large blocks).

A xapping is a collection of pairs, where the two items making up each pair are called an index and a value. Thus a xapping maps (or rather xaps) indices onto values. In the xapping $\{x \rightarrow 2\ y \rightarrow 3\ z \rightarrow 4\}$ the indices are *x*, *y*, and *z* and the values are 2, 3, and 4. The mapping from indices to values suggests that a xapping is somewhat like an association list, but in other respects it is quite different. The pairs of a xapping have no intrinsic ordering and, most importantly, they can be operated on concurrently.

Parallel operations on xappings are introduced by an operator, α, that carries the meaning "apply to all." For example, αsquare$\{x \rightarrow 2 y \rightarrow 3 z \rightarrow 4\}$ would apply the function square to the value of each pair in the xapping, yielding the new xapping $\{x \rightarrow 4 y \rightarrow 9 z \rightarrow 16\}$. Assuming that enough processors are available, all of the squarings can be done simultaneously.

Add Data Security to Your C Programs

The SECURITY LIBRARY™

Add Fast or Thorough Encryption or Compression to Your Programs WITHOUT Royalties

Build the safest and most popular methods of protecting your data directly into your program without paying royalties. Don't worry if you're not sure what to use. The discussion of security methods in the manual (with demos on disk) will help you make a choice based on your application.

With Security Library you can:

- Keep files secure on a multi-user network
- Speed up data transmission and communications through data compression
- Control access based on privileges given to a hard disk or LAN user

"I create custom software for business applications. I'm using the Security Library to encrypt so that certain information will not be readily available. The documentation is excellent. It gives a thorough explanation about how you can secure a file."

— Bruce Philips, Custom Software Design
Virginia Beach, CA

Requires MSDOS 2.0+. Works with Microsoft C and Computer Innovations' C86. Please specify compiler when you order.

You Choose the Security Level

Algorithms provided include The National Bureau of Standards' Data Encryption Standard (DES) and the Vernam and Vegenere ciphers. Encrypt a 10K file in 3 seconds with one method or 50 seconds with another.

Six algorithms are provided, along with password and non-password encryption schemes.

Valuable Extras

The Huffman coding routine can *reduce the size of a file* by 25 to 50%. The routine to change the attribute bytes of a file can make that file invulnerable to casual browsing or accidental deletion. There's even a program to *change every byte of a file to a null character* - not even un-erase programs can recover it then!

Call 800-821-2492 to order Security Library risk-free for only $125. Source & Object is $250.

Solution Systems™
335-L Washington St.,
Norwell, MA 02061
(617) 659-1571

Full refund if not satisfied in 30 days.

CIRCLE 15 ON READER SERVICE CARD

Personalize your computing environment.

The MKS Toolkit now contains the Korn shell command interpreter.

The MKS version of Bell Labs' Korn shell has this and more:

- the full power of the UNIX System V.2 Bourne shell
- the most requested features of Berkeley's C shell
- the full-UNIX utility of executable shell files
- command aliases
- interactive command-line facilities
- previous command history and editing
- a powerful programming language
- shell variable expansion
- arithmetic evaluation

All this has been fine-tuned to create the optimum environment under DOS. The Korn shell is just one of over 100 commands — fully compatible with UNIX System V.2 — now contained in the MKS Toolkit, including the following:

awk cat chmod cmp cp cpio ctags cut date
dd df diff du echo ed egrep ex fgrep
file find head help join lc ls more mv
nm od paste pg prof rm sed size sort
split strings tail time touch tr uniq vi wc

and much, much more...

These programs run from the shell or command.com under DOS on machines such as the IBM PC, XT, and AT, the AT&T 6300, and most PC compatibles. Full documentation is included. Phone support is available 9-6 EST. Not copy protected.

Everything for only $139.

Mortice Kern Systems Inc.
43 Bridgeport Road East, Waterloo, Ontario, Canada N2J 2J4
For information or ordering call collect: **(519) 884-2251**

Prices quoted in U.S. funds. MasterCard and VISA orders accepted. OEM and dealer inquiries invited. UNIX is a trademark of Bell Labs. MS-DOS is a trademark of Microsoft Corp.

CIRCLE 16 ON READER SERVICE CARD

Reflecting

In this résumé of LISP daydreams, the prize for unfettered imagination goes to Brian Cantwell Smith of the Xerox Palo Alto Research Center and Stanford University. Smith has proposed an "introspective" dialect of LISP. Its most distinctive feature is that it requires an infinite number of interpreters, all running simultaneously.[20] Furthermore, Smith and others have shown that such an ungainly monster can actually be built and made to work efficiently.

When an ordinary LISP program is being run by an interpreter there are three levels of active computation: the user program is running, the interpreter is running, and the machine that serves as host to the interpreter is running. But suppose the interpreter itself is written in LISP. Then the interpreter can start another copy of itself, which can start another copy, which can start....In this way we build an infinite tower of interpreters.

Why would anyone want to do such a thing? Smith points out that an interpreter often has information about the state of a computation that is not directly available to the interpreted program. A debugger uses such information when it displays a trace of procedure calls or lists the contents of a stack.

Many languages (including versions of LISP) provide *ad hoc* mechanisms for gaining access to the interpreter's internal knowledge. Smith has created a new language, 3-LISP, incorporating a systematic and well-structured method for allowing programs to reflect on their own execution. 3-LISP is a descendant of Scheme, with certain additional features; in particular, a procedure can be designated either *simple* or *reflect*. A *simple* procedure executes at its own level, but a *reflective* one passes its code up to the next higher level in the tower to be executed.

The trick is to avoid creating an infinite tower of interpreters. In fact, a finite tower will suffice, provided there is some level in the hierarchy above which only nonreflective procedures are being executed; the interpreters above this level will never be called on explicitly, so they can be omitted. Daniel P. Friedman of Indiana University and Mitchell Wand of Northeastern University have since shown that all the properties of the infinite reflective tower can be emulated by a single interpreter written in Scheme.[21,22]

Predicting

In nature most mutations are unhelpful, if not disastrous. However, in nature mutation is a random process, whereas the mutant LISPs described here have been carefully and deliberately designed. Even so, I feel perfectly safe in predicting that most of them will not survive, simply because there is not room enough in the world for dozens of LISP-like languages.

PLANNER and CONNIVER are already extinct. The Actors model, on the other hand, has children and grandchildren in abundance—so many, in fact, that they may suffocate their parent. Another LISP derivative, Logo, seems to have the opposite problem: Logo itself is thriving, but it may prove to be sterile. If parallel hardware ever catches on, at least one of the parallel LISPs will surely succeed. As for 3-LISP, I would not dare to guess its fate. The one sure winner among the new LISPs is Scheme, which already has a secure place in the academic world and bright prospects elsewhere.

Meanwhile, LISP itself lumbers on, although its days of carefree oat-sowing are probably over. LISP programs are no longer without economic value and the demand today is for portable, production-quality, industrial-strength compilers. The urge to standardize seems irresistable. Presumably, the standard will be Common LISP,[23] although not everyone is happy with that choice.[24] Perhaps the eventual outcome will be something like this: Common LISP will become the FORTRAN of AI, while another language—probably Scheme or a descendant of Scheme—becomes the new culture medium, the new breeding ground for linguistic innovation. ■

References

1. Steele, Guy Lewis Jr., and Gerald Jay Sussman. "The Revised Report on Scheme: A Dialect of LISP." Artifical Intelligence Memo No. 452, MIT, Cambridge, Mass., Jan. 1978.
2. Wand, Mitchell. "What Is LISP?" *American Mathematical Monthly.* (Jan. 1984):32-42.
3. Hewitt, Carl. "PLANNER: A Language for Proving Theorems in Robots." *Proceedings of the International Joint Conference on Artificial Intelligence.* (1969):295-301.
4. Kowalski, Robert. "The Origins of Logic Programming." *Byte.* (Aug. 1985):192-193.
5. Robinson, J. A., and E. E. Sibert. "LOGLISP: Motivation, Design and Implementation." *Logic Programming.* Academic Press, 1982, pp. 299-313.
6. Kornfeld, William A. "Pattern-Directed Invocation Languages." *Byte.* (Aug. 1979):34-48.
7. Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. "MicroPLANNER Reference Manual." Artificial Intelligence Memo No. 203A, MIT, Cambridge, Mass., 1971.
8. Sussman, Gerald Jay, and Drew V. McDermott. "Why Conniving Is Better Than Planning." Artificial Intelligence Memo No. 255A, MIT, Cambridge, Mass., 1972.
9. Hewitt, Carl. "Control Structure as Patterns of Passing Messages." *Artificial Intelligence.* vol. 8, no. 3 (June 1977):323-363. Also published in *Artificial Intelligence: An MIT Perspective.* vol. 2:435-465, The MIT Press, 1979.
10. Pugh, John R. "Actors—The Stage is Set." *SIGPLAN Notices.* vol. 19, no. 3 (Mar. 1984):61-65.
11. Hewitt, Carl. "The Apiary Network Architecture for Knowledgeable Systems." *Conference Record of the 1980 LISP Conference.* Association for Computing Machinery (reprinted). (1985):107-117.
12. Bobrow, Daniel G., and M. J. Stefik. *The LOOPS Manual.* Xerox Corp. 1983.
13. Weinreb, Daniel and David Moon. "Flavors: Message Passing in the Lisp Machine." Artificial Intelligence Memo No. 602, MIT, Cambridge, Mass., Nov. 1980.
14. Betz, David. "An XLISP Tutorial." *Byte.* (Mar. 1985):221-236.
15. Sussman, Gerald Jay, and Guy Lewis Steele Jr. "Scheme: An Interpreter for Extended Lambda Calculus." Artificial Intelligence Memo No. 349, MIT, Cambridge, Mass., Dec. 1975.
16. Rees, Jonathan, and William Clinger, ed. "Revised[3] Report on the Algorithmic Language Scheme." Artificial Intelligence Memo No. 848a, MIT, Cambridge, Mass., Sept. 1986.
17. Gelernter, David, Suresh Jagannathan, and Thomas London. "Environments as First Class Objects." *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages.* Jan. 1987 (in press).
18. Halstead, Robert H. Jr. "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems.* vol. 7, no. 4 (Oct. 1985):501-538.
19. Steele, Guy L. Jr. and W. Daniel Hillis. "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing." *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.* (Aug. 1986):279-297.
20. Des Rivières, Jim, and Brian Cantwell Smith. "The Implementation of Procedurally Reflective Languages." *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* (Aug. 1984):331-347.
21. Friedman, Daniel P., and Mitchell Wand. "Reification: Reflection without Metaphysics." *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.*(Aug. 1984):348-355.
22. Wand, Mitchell, and Daniel P. Friedman. "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower." *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* (Aug. 1986):298-307.
23. Steele, Guy L. Jr., et al. *Common LISP: The Language.* Digital Press, 1984.
24. Allen, John R. "The Death of Creativity: Is Common LISP a LISP-like Language?" *AI Expert.* vol. 2, no. 2. (Feb. 1987):48-61.