# LISP—NOTES ON ITS PAST AND FUTURE—1980

**John McCarthy**
Computer Science Department
Stanford University
Stanford, CA 94305
jmc@cs.stanford.edu
http://www-formal.stanford.edu/jmc/

1999 Mar 22, 5:09 p.m.

**Abstract**

LISP has survived for 21 years because it is an approximate local optimum in the space of programming languages. However, it has accumulated some barnacles that should be scraped off, and some long-standing opportunities for improvement have been neglected. It would benefit from some co-operative maintenance especially in creating and maintaining program libraries. Computer checked proofs of program correctness are now possible for pure LISP and some extensions, but more theory and some smoothing of the language itself are required before we can take full advantage of LISP's mathematical basis.

1999 note: This article was included in the 1980 Lisp conference held at Stanford. Since it almost entirely corresponds to my present opinions, I should have asked to have it reprinted in the 1998 Lisp users conference proceedings at which I gave a talk with the same title.

# 1 Introduction

On LISP's approximate 21st anniversary, no doubt something could be said about coming of age, but it seems doubtful that the normal life expectancy of a programming language is three score and ten. In fact, LISP seems to be the second oldest surviving programming language after Fortran, so maybe we should plan on holding one of these newspaper interviews in which grandpa is asked to what he attributes having lived to 100. Anyway the early history of LISP was already covered in [McC81], reprinted from the Proceedings of the 1977 ACM conference on the history of programming languages.

Therefore, these notes first review some of the salient features of LISP and their relation to its long survival, noting that it has never been supported by a computer company. LISP has a partially justified reputation of being more based on theory than most computer languages, presumably stemming from its functional form, its use of lambda notation and basing the interpreter on a universal function.

From the beginning, I have wanted to develop techniques for making computer checkable proofs of LISP programs, and now this is possible for a large part of LISP. Still other present and proposed facilities are in a theoretically more mysterious state. I will conclude with some remarks on improvements that might be made in LISP and the prospects for replacing it by something substantially better.

# 2 The Survival of LISP

As a programming language, LISP is characterized by the following ideas:

1. Computing with symbolic expressions rather than numbers.
2. Representation of symbolic expressions and other information by list structure in computer memory.
3. Representation of information on paper, from keyboards and in other external media mostly by multi-level lists and sometimes by S-expressions. It has been important that any kind of data can be represented by a single general type.
4. A small set of selector and constructor operations expressed as functions, i.e. *car*, *cdr* and *cons*.

5. Composition of functions as a tool for forming more complex functions.

6. The use of conditional expressions for getting branching into function definitions.

7. The recursive use of conditional expressions as a sufficient tool for building computable functions.

8. The use of $\lambda$-expressions for naming functions.

9. The storage of information on the property lists of atoms.

10. The representation of LISP programs as LISP data that can be manipulated by object programs. This has prevented the separation between system programmers and application programmers. Everyone can "improve" his LISP, and many of these "improvements" have developed into improvements to the language.

11. The conditional expression interpretation of Boolean connectives.

12. The LISP function *eval* that serves both as a formal definition of the language and as an interpreter.

13. Garbage collection as the means of erasure.

14. Minimal requirements for declarations so that LISP statements can be executed in an on-line environment without preliminaries.

15. LISP statements as a command language in an on-line environment.

Of course, the above doesn't mention features that LISP has in common with most programming languages in its "program feature".

All these features have remained viable and the combination must be some kind of approximate local optimum in the space of programming languages, because LISP has survived several attempts to replace it, some rather determined. It may be worthwhile to review a few of these and guess why they didn't make it.

1. SLIP included list processing in Fortran. It used bidirectional lists and didn't allow recursive functions or conditional expressions. The bidirectional lists offered advantages in only a few applications but otherwise took up space and time. It didn't encourage on-line use, since Fortran doesn't.

2. Formac was another Fortran based language that was pushed for a while by part of IBM. It was dedicated to manipulating a class of algebraic formulas written in Fortran style and was also oriented to batch processing.

3. Formula Algol was dedicated to the linguistic pun that the elementary operations can be regarded as operating on numbers or on formulas. The idea was that if a variable $x$ has no value, then operations on expressions involving $x$ must be regarded as operating on the formula. A few programs could be written, but the pun proved an inadequate basis for substantial programs.

4. One of the more interesting rivals to LISP is (or was) POP-2. It has everything that LISP has except that its statements are written in an Algol-like form and don't have any list structure internal form. Thus POP-2 programs can produce other POP-2 programs only as character strings. This makes a much sharper distinction between system programmers and application programmers than in LISP. In LISP, for example, anyone can make his own fancy macro recognizer and expander.

5. Microplanner is an attempt to make a higher level general purpose language than LISP. The higher level involves both data (pattern matching) and control (goal seeking and failure mechanisms). Unfortunately, both proved inadequately general, and programmers were forced to very elaborate constructions, to new languages like CONNIVER with even more elaborate control features, and eventually many went back to LISP.

   One generic trouble seems to be that no-one adequately understands pattern directed computation which always works very nicely on simple examples, but which leads to over complicated systems when generalized. We can see this in LISP in certain macro expansion systems like that of the LISP machine [WM78].

6. I should mention Prolog, but I don't understand it well enough to comment. [1]

[1] 1999 note: The ideas of Prolog are similar to a subset of the ideas of Microplanner. However, Prolog was designed systematically and has survived, while Microplanner didn't.

# 3 Improvements

Like most everything, LISP is subject to improvement. The various versions of LISP have accumulated many barnacles with time, and these would have to be scraped off before a definitive standardizable language could be achieved - a worthwhile but long term goal. Meanwhile here are a few directions for improvement. Some are purely operational and others have more conceptual content.

1. Incorporating more standard functions into the language and rationalizing the standard functions in the present versions.

   Designers of programming languages often propose omitting from the definition of the language facilities that can be defined within the language on the grounds that the user can do it for himself. The result is often that users cannot use each other's programs, because each installation and user performs various common tasks in different ways. In so far as programmers use local libraries without rewriting the functions, they are using different languages if they use different local libraries. Compatibility between users of LISP would be much enhanced if there were more standard functions.

2. Syntax directed input and output.

   A notation for representing symbolic information can be optimized from three points of view: One can try to make it easy to write. One can try to make it easy and pleasant to read. One can make easy to manipulate with computer programs. Unfortunately, these desiderata are almost always grossly incompatible. LISP owes most of its success to optimizing the third. LISP lists and S-expressions in which the *car* of an item identifies its kind have proved most suitable as data for programming. When the amount of input and output is small, users are inclined to accept the inconvenience of entering the input and seeing the output as lists or S-expressions. Otherwise they write *read* and *print* programs of varying elaborateness. Input and output programs are often a large part of the work and a major source of bugs. Moreover, input programs often must detect and report errors in the syntax of input.

---

The key permanent idea of Prolog is that a certain subset of logic (Horn clauses) are executable as programs doing backtracking search. It seems to me that this discovery has as much permanent importance as the ideas behind Lisp.

LISP would be much improved by standard facilities for syntax directed input and output. Some years ago Lynn Quam implemented a system that used the same syntax description for both input and output, but this was rather constraining. Probably one wants different syntaxes for input and output, and input syntaxes should specify ways of complaining about errors. The idea is to provide standard facilities for a programmer to describe correspondences between data in an external medium and S-expressions, e.g. he should be able to say something like

$(PLUS\ x \ldots z) \to x + \ldots + z,$

$(DIFFERENCE\ x\ y) \to x - y,$

although I hold no brief for this particular notation.

3. Syntax directed computation in general.

   It isn't clear whether this would be a feature to be added to LISP or a new language. However, it seems likely that both the functional form of computation that LISP has now and syntax directed features are wanted in one language.

4. LISP might benefit if we could find a way to finance and manage a central agency that could keep libraries, make agreed on machine independent improvements, maintain a standard subset, and co-ordinate pressure on computer manufacturers to develop and maintain adequate LISPs on their machines. It shouldn't get too powerful.[2]

# 4 Proving Correctness of LISP Programs

This can be done by taking Advantage of LISP's Theoretical Foundation.

As soon as pure LISP took its present form, it became apparent that properties of LISP functions should be provable by algebraic manipulation together with an appropriate form of mathematical induction. This gave rise to the goal of creating a mathematical theory of computation that would lead to computer checked proofs [McC62] that programs meet their specifications. Because LISP functions are functions, standard logical techniques weren't immediately applicable, although recursion induction [McC63] quickly became available as an informal method. The methods of [Kle52] might have been

[2]1999: Only the part about standardization happened.

adopted to proving properties of programs had anyone who understood them well been properly motivated and understood the connections.

The first adequate formal method was based on Cartwright's thesis [Car77], which permits a LISP function definition such as

$$\begin{aligned} &append[u,v] \leftarrow \textbf{if null } u \textbf{ then } v \\ &\quad \textbf{else } cons[car\ u, append[cdr\ u, v]] \end{aligned}$$

to be replaced by a first order sentence

$$\begin{aligned} &(\forall u\ v)(append(u,v) = \textbf{if null } u \textbf{ then } v \\ &\quad \textbf{else } cons(car\ u, append(cdr\ u, v))) \end{aligned}$$

without first having to prove that the program terminates for any lists $u$ and $v$. The proof of termination has exactly the same form as any other inductive proof. See also [CM79].

The Elephant formalism (McCarthy 1981 forthcoming)[3] supplies a second method appropriate for sequential LISP programs. Boyer and Moore [BM79] provide proof finding as well as proof checking in a different formalism that requires a proof that a function is total as part of the process of accepting its definition.

I should say that I don't regard the LCF methods as adequate, because the "logic of computable functions" is too weak to fully specify programs.

These methods (used informally) have been succesfully taught as part of the LISP course at Stanford and will be described in the textbook (McCarthy and Talcott 1980).[4] It is also quite feasible to check the proofs by machine using Richard Weyhrauch's FOL interactive proof-checker for first order logic, but practical use requires a LISP system that integrates the proof checker with the interpreter and compiler.[5][6]

[3] 1999: The 1981 ideas have been combined with other ideas, e.g. about speech acts, and elaborated. See [McC96]. The Elephant idea referred to was to avoid data structures by allowing direct reference to the past.

[4] 1999: That textbook didn't appear, mainly because of a difference of opinion among the authors about the most appropriate proof formalism

[5] 1999: FOL was succeeded in the Lisp course by Jussi Ketonen's EKL prover, but the proposed integrated system hasn't happened.

[6] 1999: NQTHM (aka the Boyer-Moore prover) was used by Shankar when he taught the course. This prover is designed to use induction to prove properties of total Lisp functions. Using the Eval function of the logic and the representation of function definitions as Sexpressions properties of partial functions can also be proved. NQTHM has evolved

The ultimate goal of computer proof-checking is a system that will be used by people without mathematical inclination simply because it leads more quickly to programs without bugs. This requires further advances that will make possible shorter proofs and also progress in writing the specifications of programs.

Probably some parts of the specifications such as that the program terminates are almost syntactic in their checkability. However, the specifications of programs used in AI work require new ideas even to formulate. I think that recent work in non-monotonic reasoning will be relevant here, because the fact that an AI program works requires jumping to conclusions about the world in which it operates.

While pure LISP and the simple form of the "program feature" are readily formalized, many of the fancier features of the operational LISP systems such as Interlisp, Maclisp and Lisp Machine LISP [WM78] are harder to formalize. Some of them like FEXPRs require more mathematical research, but others seem to me to be kludges and should be made more mathematically neat both so that properties of programs that use them can be readily proved and also to reduce ordinary bugs.

The following features of present LISP systems and proposed extensions require new methods for correctness proofs:

1. Programs that involve re-entrant list structure. Those that don't involve *rplaca* and *rplacd* such as search and print programs are more accessible than those that do. I have an induction method on finite graphs that applies to them, but I don't yet know how to treat *rplaca*, etc. Induction on finite graphs also has applications to proving theorems about flowchart programs.[7]

2. No systematic methods are known for formally stating and proving properties of syntax directed computations.[8]

3. Programs that use macro expansions are in principle doable via axiomatizations of the interpreter, but I don't know of any actual formal

into ACL2 which supports a large applicative subset of Common Lisp and is programmed almost entirely within that subset. [see http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html]

[7] 1999: Ian Mason's thesis[Mas86] gave some principles for reasoning about first-order Lisp including rplacx.

[8] 1999: Maybe this is still the case.

proofs.

4. No techniques exist for correctness proofs of programs involving lazy evaluators.

5. Programs with functional arguments are in principle accessible by Dana Scott's methods, but the different kinds of functional arguments have been treated only descriptively and informally.[9]

6. Probably the greatest obstacle to making proof-checking a useful tool is our lack of knowledge of how to express the specifications of programs. Many programs have useful partial specifications - they shouldn't loop or modify storage that doesn't belong to them. A few satisfy algebraic relations, and this includes compilers. However, programs that interact with the world have specifications that involve assumptions about the world. AI programs in general are difficult to specify; most likely their very specification involves default and other non-monotonic reasoning. (See [McC80].)

# 5 Mysteries and other Matters

1. Daniel Friedman and David Wise have argued that *cons* should not evaluate its arguments and have shown that this allows certain infinite list structures to be regarded as objects. Trouble is avoided, because only as much of the infinite structure is created as is necessary to get the answers to be printed. Exactly what domain of infinite list structures is assumed is unclear to me. While they give interesting examples of applications, it isn't clear whether the proposed extension has practical value.

2. Many people have proposed implementations of full lambda calculus. This permits higher level functions, i.e. functions of functions of functions etc., but allows only manipulations based on composition and lambda conversions, not general manipulations of the symbolic form of

[9] 1999: A logic for reasoning about Lisp-Scheme-ML like programs with functions and mutable data structures has been developed by Mason and Talcott [HMST95]. This logic has a relatively complete axiomatization of primitives for mutable data as well as a variety of induction principles and methods for proving properties of programs.

functions. While conditional expressions are not directly provided, they can be imitated by writing (as proposed by Dana Scott in an unpublished note) **true** as $(\lambda x\ y.x)$, **false** as $(\lambda x\ y.y)$ and **if** $p$ **then** $a$ **else** $b$ as $p(a)(b)$. Another neat idea of Scott's (improved from one of Church) is to identify the natural number $n$ with the operation of taking the (n+1)th element of a list. The mystery is whether extension to lambda calculus has any practical significance, and the current best guess is no, although the Scott's notational idea suggests changing the notation of LISP and writing 0 for *car*, 1 for *cadr*, 2 for *caddr*, etc.

3. Pure LISP would be much simpler conceptually if all list structure were represented uniquely in memory. This can be done using a hash *cons*, but then *rplaca* and friends don't work. Can't we somehow have the best of both worlds?

4. It seems to me that LISP will probably be superseded for many purposes by a language that does to LISP what LISP does to machine language. Namely it will be a higher level language than LISP that, like LISP and machine language, can refer to its own programs. (However, a higher level language than LISP might have such a large declarative component that its texts may not correspond to programs. If what replaces the interpreter is smart enough, then the text written by a user will be more like a declarative description of the facts about a goal and the means available for attaining it than a program per se).[10]

   An immediate problem is that both the kinds of abstract syntax presently available and present pattern matching systems are awkward for manipulating expressions containing bound variables.

# 6 References

Lisp was first described in [McC60] and the first manual was [ML+66] the first version of which appeared in 1962.[11]

[10] 1999: An example is Maude [Gro99, CDE+98, Wil97], a language based on Rewriting Logic. In Maude, actions and effects are expressed in a declarative manner, and using the reflective capability, Maude programs and computations can be represented, reasoned about, modified and executed in Maude.

[11] 1999: I thank Carolyn Talcott for additonal references.

## References


[BM79] Robert Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[Car77] Robert S. Cartwright. A practical formal semantic definition and verification system for typed lisp. Phd dissertation, stanford university, 1977.

[CDE+98] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, and J. Meseguer. Metalevel Computation in Maude. In C. Kirchner and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998. URL: `http://www.elsevier.nl/locate/entcs/volume15.html`.

[CM79] Robert Cartwright and John McCarthy. Recursive programs as functions in a first order theory. In *Proceedings of the International Conference on Mathematical Studies of Information Processing, Kyoto, Japan*, 1979.

[Gro99] The Maude Group. The Maude system, 1999. See http://maude.csl.sri.com/.

[HMST95] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1952.

[Mas86] I. A. Mason. *The Semantics of Destructive Lisp.* PhD thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Comm. A.C.M.*, 3:184–195, 1960.

[McC62] John McCarthy. Checking mathematical proofs by computer. In *Proceedings Symposium on Recursive Function Theory (1961)*. American Mathematical Society, 1962.

[McC63] John McCarthy. A Basis for a Mathematical Theory of Computation[12]. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[McC80] John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning[13]. *Artificial Intelligence*, 13:27–39, 1980. Reprinted in [McC90].

[McC81] John McCarthy. History of lisp. In Richard L. Wexelblat, editor, *History of programming languages*. Academic Press, 1981. Reprinted from Proceedings of the ACM Conference on the History of Programming Languages, Los Angeles, 1977.

[McC90] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy*. Ablex Publishing Corporation, 355 Chestnut Street, Norwood, NJ 07648, 1990.

[McC96] John McCarthy. **elephant 2000**[14]. Technical report, Stanford Formal Reasoning Group, 1996. Available only as http://www-formal.stanford.edu/jmc/elephant.html.

[ML+66] John McCarthy, Michael Levin, et al. *LISP 1.5 Programmer's Manual*. MIT, 1966.

[Wil97] Wilson97. Cars and their enemies. *Commentary*, pages 17–23, July 1997.

[WM78] Daniel Weinreb and David Moon. Lisp machine manual. Technical report, M.I.T. Artificial Intelligence Laboratory, 1978.

/@steam.stanford.edu:/u/ftp/jmc/lisp20th.tex: begun Mon Feb 1 17:36:27 1999, latexed March 22, 1999 at 5:09 p.m.

[12]http://www-formal.stanford.edu/jmc/basis.html

[13]http://www-formal.stanford.edu/jmc/circumscription.html

[14]http://www-formal.stanford.edu/jmc/elephant.html