History of Lisp

John McCarthy
Artificial Intelligence Laboratory
Stanford University

12 February 1979

This draft gives insufficient mention to many people who helped implement LISP and who contributed ideas. Suggestions for improvements in that directions are particularly welcome. Facts about the history of FUNARG and uplevel addressing generally are especially needed.

1 Introduction

This paper concentrates on the development of the basic ideas and distinguishes two periods - Summer 1956 through Summer 1958 when most of the key ideas were developed (some of which were implemented in the FORTRAN based FLPL), and Fall 1958 through 1962 when the programming language was implemented and applied to problems of artificial intelligence. After 1962, the development of LISP became multi-stranded, and different ideas were pursued in different places.

Except where I give credit to someone else for an idea or decision, I should be regarded as tentatively claiming credit for it or else regarding it as a consequence of previous decisions. However, I have made mistakes about such matters in the past, and I have received very little response to requests for comments on drafts of this paper. It is particularly easy to take as obvious a feature that cost someone else considerable thought long ago. As the writing of this paper approaches its conclusion, I have become aware of additional sources of information and additional areas of uncertainty.

As a programming language, LISP is characterized by the following ideas: computing with symbolic expressions rather than numbers, representation of symbolic expressions and other information by list structure in the memory of a computer, representation of information in external media mostly by multi-level lists and sometimes by S-expressions, a small set of selector and constructor operations expressed as functions, composition of functions as a tool for forming more complex functions, the use of conditional expressions for getting branching into function definitions, the recursive use of conditional expressions as a sufficient tool for building computable functions, the use of λ-expressions for naming functions, the representation of LISP programs as LISP data, the conditional expression interpretation of Boolean connectives, the LISP function *eval* that serves both as a formal definition of the language and as an interpreter, and garbage collection as a means of handling the erasure problem. LISP statements are also used as a command language when LISP is used in a time-sharing environment.

Some of these ideas were taken from other languages, but most were new. Towards the end of the initial period, it became clear that this combination of ideas made an elegant mathematical system as well as a practical programming language. Then mathematical neatness became a goal and led to pruning some features from the core of the language. This was partly motivated by esthetic reasons and partly by the belief that it would be easier to devise techniques for proving programs correct if the semantics were compact and without exceptions. The results of (Cartwright 1976) and (Cartwright and McCarthy 1978), which show that LISP programs can be interpreted as sentences and schemata of first order logic, provide new confirmation of the original intuition that logical neatness would pay off.

2 LISP prehistory - Summer 1956 through Summer 1958.

My desire for an algebraic list processing language for artificial intelligence work on the IBM 704 computer arose in the summer of 1956 during the Dartmouth Summer Research Project on Artificial Intelligence which was the first organized study of AI. During this meeting, Newell, Shaw and Simon described IPL 2, a list processing language for Rand Corporation's JOHNNIAC computer in which they implemented their Logic Theorist program. There was little temptation to copy IPL, because its form was based on a JOHNNIAC loader that happened to be available to them, and because the

FORTRAN idea of writing programs algebraically was attractive. It was immediately apparent that arbitrary subexpressions of symbolic expressions could be obtained by composing the functions that extract immediate subexpressions, and this seemed reason enough to go to an algebraic language.

There were two motivations for developing a language for the IBM 704. First, IBM was generously establishing a New England Computation Center at M.I.T. which Dartmouth would use. Second, IBM was undertaking to develop a program for proving theorems in plane geometry (based on an idea of Marvin Minsky's), and I was to serve as a consultant to that project. At the time, IBM looked like a good bet to pursue artificial intelligence research vigorously, and further projects were expected. It was not then clear whether IBM's FORTRAN project would lead to a language within which list processing could conveniently be carried out or whether a new language would be required. However, many considerations were independent of how that might turn out.

Apart from consulting on the geometry program, my own research in artificial intelligence was proceeding along the lines that led to the Advice Taker proposal in 1958 (McCarthy 1959). This involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate - it still is - and a list processing language also seemed appropriate for programming the operations involved in deduction - and still is.

This internal representation of symbolic information gives up the familiar infix notations in favor of a notation that simplifies the task of programming the substantive computations, e.g. logical deduction or algebraic simplification, differentiation or integration. If customary notations are to be used externally, translation programs must be written. Thus most LISP programs use a prefix notation for algebraic expressions, because they usually must determine the main connective before deciding what to do next. In this LISP differs from almost every other symbolic computation system. COMIT, FORMAC, and Formula Algol programs all express the computations as operations on some approximation to the customary printed forms of symbolic expressions. SNOBOL operates on character strings but is neutral on how character strings are used to represent symbolic information. This feature probably accounts for LISP's success in competition with these languages, especially when large programs have to be written. The advantage is like that of binary computers over decimal - but larger.

(In the late 1950s, neat output and convenient input notation was not generally considered important. Programs to do the kind of input and output customary today wouldn't even fit in the memories available at that time. Moreover, keypunches and printers with adequate character sets didn't exist).

The first problem was how to do list structure in the IBM 704. This computer has a 36 bit word, and two 15 bit parts, called the address and decrement, were distinguished by special instructions for moving their contents to and from the 15 bit index registers. The address of the machine was 15 bits, so it was clear that list structure should use 15 bit pointers. Therefore, it was natural to consider the word as divided into 4 parts, the address part, the decrement part, the prefix part and the tag part. The last two were three bits each and separated from each other by the decrement so that they could not be easily combined into a single six bit part.

At this point there was some indecision about what the basic operators should be, because the operation of extracting a part of the word by masking was considered separately from the operation of taking the contents of a word in memory as a function of its address. At the time, it seemed dubious to regard the latter operation as a function, since its value depended on the contents of memory at the time the operation was performed, so it didn't act like a proper mathematical function. However, the advantages of treating it grammatically as a function so that it could be composed were also apparent.

Therefore, the initially proposed set of functions included *cwr*, standing for "Contents of the Word in Register number" and four functions that extracted the parts of the word and shifted them to a standard position at the right of the word. An additional function of three arguments that would also extract an arbitrary bit sequence was also proposed.

It was soon noticed that extraction of a subexpression involved composing the extraction of the address part with *cwr* and that continuing along the list involved composing the extraction of the decrement part with *cwr*. Therefore, the compounds *car*, standing for "Contents of the Address part of Register number", and its analogs *cdr*, *cpr*, and *ctr* were defined. The motivation for implementing *car* and *cdr* separately was strengthened by the vulgar fact that the IBM 704 had instructions (connected with indexing) that made these operations easy to implement. A construct operation for taking a word off the free storage list and stuffing it with given contents was also obviously required. At some point a *cons(a,d,p,t)* was defined, but it was regarded as a subroutine and not as a function with a value. This work was done at Dartmouth, but not on a computer, since the New England Computation

Center was not expected to receive its IBM 704 for another year.

In connection with IBM's plane geometry project, Nathaniel Rochester and Herbert Gelernter (on the advice of McCarthy) decided to implement a list processing language within FORTRAN, because this seemed to the the easiest way to get started, and, in those days, writing a compiler for a new language was believed to take many man-years. This work was undertaken by Herbert Gelernter and Carl Gerberich at IBM and led to FLPL, standing for FORTRAN List Processing Language. Gelernter and Gerberich noticed that *cons* should be a function, not just a subroutine, and that its value should be the location of the word that had been taken from the free storage list. This permitted new expressions to be constructed out of subsubexpressions by composing occurrences of *cons*.

While expressions could be handled easily in FLPL, and it was used successfully for the Geometry program, it had neither conditional expressions nor recursion, and erasing list structure was handled explicitly by the program.

I invented conditional expressions in connection with a set of chess legal move routines I wrote in FORTRAN for the IBM 704 at M.I.T. during 1957-58. This program did not use list processing. The IF statement provided in FORTRAN 1 and FORTRAN 2 was very awkward to use, and it was natural to invent a function XIF(M,N1,N2) whose value was N1 or N2 according to whether the expression M was zero or not. The function shortened many programs and made them easier to understand, but it had to be used sparingly, because all three arguments had to be evaluated before XIF was entered, since XIF was called as an ordinary FORTRAN function though written in machine language. This led to the invention of the true conditional expression which evaluates only one of N1 and N2 according to whether M is true or false and to a desire for a programming language that would allow its use.

A paper defining conditional expressions and proposing their use in Algol was sent to the *Communications of the ACM* but was arbitrarily demoted to a letter to the editor, because it was very short.

I spent the summer of 1958 at the IBM Information Research Department at the invitation of Nathaniel Rochester and chose differentiating algebraic expressions as a sample problem. It led to the following innovations beyond FLPL:

a. Writing recursive function definitions using conditional expressions. The idea of differentiation is obviously recursive, and conditional expressions allowed combining the cases into a single formula.

b. The *maplist* function that forms a list of applications of a functional argument to the elements of a list. This was obviously wanted for differentiating sums of arbitrarily many terms, and with a slight modification, it could be applied to differentiating products. (The original form was what is now called *mapcar*).

c. To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn't understand the rest of his book, so I wasn't tempted to try to implement his more general mechanism for defining functions. Church used higher order functionals instead of using conditional expressions. Conditional expressions are much more readily implemented on computers.

d. The recursive definition of differentiation made no provision for erasure of abandoned list structure. No solution was apparent at the time, but the idea of complicating the elegant definition of differentiation with explicit erasure was unattractive. Needless to say, the point of the exercise was not the differentiation program itself, several of which had already been written, but rather clarification of the operations involved in symbolic computation.

In fact, the differentiation program was not implemented that summer, because FLPL allows neither conditional expressions nor recursive use of subroutines. At this point a new language was necessary, since it was very difficult both technically and politically to tinker with Fortran, and neither conditional expressions nor recursion could be implemented with machine language Fortran functions - not even with "functions" that modify the code that calls them. Moreover, the IBM group seemed satisfied with FLPL as it was and did not want to make the vaguely stated but obviously drastic changes required to allow conditional expressions and recursive definition. As I recall, they argued that these were unnecessary.

3 The implementation of LISP

In the Fall of 1958, I became Assistant Professor of Communication Sciences (in the EE Department) at M.I.T., and Marvin Minsky (then an assistant professor in the Mathematics Department) and I started the M.I.T. Artificial Intelligence Project. The Project was supported by the M.I.T. Research Laboratory of Electronics which had a contract from the armed services that permitted great freedom to the Director, Professor Jerome Wiesner, in initiating new projects that seemed to him of scientific interest. No written proposal was ever made. When Wiesner asked Minsky and me what we needed for the project, we asked for a room, two programmers, a secretary and a keypunch, and he asked us to also undertake the supervision of some of the six mathematics graduate students that R.L.E. had undertaken to support.

The implementation of LISP began in Fall 1958. The original idea was to produce a compiler, but this was considered a major undertaking, and we needed some experimenting in order to get good conventions for subroutine linking, stack handling and erasure. Therefore, we started by hand-compiling various functions into assembly language and writing subroutines to provide a LISP "environment". These included programs to read and print list structure. I can't now remember whether the decision to use parenthesized list notation as the external form of LISP data was made then or whether it had already been used in discussing the paper differentiation program.

The programs to be hand-compiled were written in an informal notation called M-expressions intended to resemble FORTRAN as much as possible. Besides FORTRAN-like assignment statements and **go to**s, the language allowed conditional expressions and the basic functions of LISP. Allowing recursive function definitions required no new notation from the function definitions allowed in FORTRAN I - only the removal of the restriction - as I recall, unstated in the FORTRAN manual - forbidding recursive definitions. The M-notation also used brackets instead of parentheses to enclose the arguments of functions in order to reserve parentheses for list-structure constants. It was intended to compile from some approximation to the M-notation, but the M-notation was never fully defined, because representing LISP functions by LISP lists became the dominant programming language when the interpreter later became available. A machine readable M-notation would have required redefinition, because the pencil-and-paper M-notation used characters unavailable on the IBM 026 key punch.

The READ and PRINT programs induced a *de facto* standard external notation for symbolic information, e.g. representing $x + 3y + z$ by (PLUS X (TIMES 3 Y) Z) and $(\forall x)(P(x) \vee Q(x,y)$ by (ALL (X) (OR (P X) (Q X Y))). Any other notation necessarily requires special programming, because standard mathematical notations treat different operators in syntactically different ways. This notation later came to be called "Cambridge Polish", because it resembled the prefix notation of Lukasiewicz, and because we noticed that Quine had also used a parenthesized prefix notation.

The erasure problem also had to be considered, and it was clearly unaesthetic to use explicit erasure as did IPL. There were two alternatives. The first was to erase the old contents of a program variable whenever it was updated. Since the *car* and *cdr* operations were not to copy structure, merging list structure would occur, and erasure would require a system of reference counts. Since there were only six bits left in a word, and these were in separated parts of the word, reference counts seemed infeasible without a drastic change in the way list structures were represented. (A list handling scheme using reference counts was later used by Collins (1960) on a 48 bit CDC computer).

The second alternative is *garbage collection* in which storage is abandoned until the free storage list is exhausted, the storage accessible from program variables and the stack is marked, and the unmarked storage is made into a new free storage list. Once we decided on garbage collection, its actual implementation could be postponed, because only toy examples were being done.

At that time it was also decided to use SAVE and UNSAVE routines that use a single contiguous public stack array to save the values of variables and subroutine return addresses in the implementation of recursive subroutines. IPL built stacks as list structure and their use had to be explicitly programmed. Another decision was to give up the prefix and tag parts of the word, to abandon *cwr*, and to make *cons* a function of two arguments. This left us with only a single type - the 15 bit address - so that the language didn't require declarations.

These simplifications made LISP into a way of describing computable functions much neater than the Turing machines or the general recursive definitions used in recursive function theory. The fact that Turing machines constitute an awkward programming language doesn't much bother recursive function theorists, because they almost never have any reason to write particular recursive definitions, since the theory concerns recursive functions in general. They often have reason to prove that recursive functions with specific properties exist, but this can be done by an informal argument without having to write them down explicitly. In the early days of computing, some people developed programming languages based on Turing machines; perhaps it seemed more scientific. Anyway, I decided to write a paper describing LISP both as a programming language and as a formalism for doing recursive function theory. The paper was *Recursive functions of symbolic expressions and their computation by machine, part I* (McCarthy 1960). Part

II was never written but was intended to contain applications to computing with algebraic expressions. The paper had no influence on recursive function theorists, because it didn't address the questions that interested them.

One mathematical consideration that influenced LISP was to express programs as applicative expressions built up from variables and constants using functions. I considered it important to make these expressions obey the usual mathematical laws allowing replacement of expressions by expressions giving the same value. The motive was to allow proofs of properties of programs using ordinary mathematical methods. This is only possible to the extent that side-effects can be avoided. Unfortunately, side-effects are often a great convenience when computational efficiency is important, and "functions" with side-effects are present in LISP. However, the so-called pure LISP is free of side-effects, and (Cartwright 1976) and (Cartwright and McCarthy 1978) show how to represent pure LISP programs by sentences and schemata in first order logic and prove their properties. This is an additional vindication of the striving for mathematical neatness, because it is now easier to prove that pure LISP programs meet their specifications than it is for any other programming language in extensive use. (Fans of other programming languages are challenged to write a program to concatenate lists and prove that the operation is associative).

Another way to show that LISP was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the LISP function *eval[e,a]*, which computes the value of a LISP expression e - the second argument a being a list of assignments of values to variables. (a is needed to make the recursion work). Writing *eval* required inventing a notation representing LISP functions as LISP data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express LISP programs in practice. Logical completeness required that the notation used to express functions used as functional arguments be extended to provide for recursive functions, and the LABEL notation was invented by Nathaniel Rochester for that purpose. D.M.R. Park pointed out that LABEL was logically unnecessary since the result could be achieved using only LAMBDA - by a construction analogous to Church's Y-operator, albeit in a more complicated way.

S.R. Russell noticed that *eval* could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.

The unexpected appearance of an interpreter tended to freeze the form of the language, and some of the decisions made rather lightheartedly for the "Recursive functions ..." paper later proved unfortunate. These included the COND notation for conditional expressions which leads to an unnecessary depth of parentheses, and the use of the number zero to denote the empty list NIL and the truth value **false**. Besides encouraging pornographic programming, giving a special interpretation to the address 0 has caused difficulties in all subsequent implementations.

Another reason for the initial acceptance of awkwardnesses in the internal form of LISP is that we still expected to switch to writing programs as M-expressions. The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised.

4 From LISP 1 to LISP 1.5

a. Property lists. The idea of providing each atom with a list of properties was present in the first assembly language implementation. It was also one of the theoretical ideas of the Advice Taker, although the Advice Taker (McCarthy 1959) would have required a property list for any expression about which information was known that did not follow from its structure. The READ and PRINT programs required that the print names of atoms be accessible, and as soon as function definition became possible, it was necessary to indicate whether a function was a SUBR in machine code or was an EXPR represented by list structure. Several functions dealing with property lists were also made available for application programs which made heavy use of them.

b. Insertion of elements in lists and their deletion. One of the original advertised virtues of list processing for AI work was the ability to insert and delete elements of lists. Unfortunately, this facility coexists uneasily with shared list structure. Moreover, operations that insert and delete don't have a neat representation as functions. LISP has them in the form of the *rplaca* and *rplacd* pseudo-functions, but programs that use them cannot be conveniently represented in logic, because, regarded as functions, they don't permit replacement of equals by equals.

c. Numbers. Many computations require both numbers and symbolic expressions. Numbers were originally implemented in LISP I as lists of atoms, and this proved too slow for all but the simplest computations. A reasonably efficient implementation of numbers as atoms in S-expressions was made in LISP 1.5, but in all the early LISPs, numerical computations were still 10 to 100 times slower than in FORTRAN. Efficient numerical computation requires some form of typing in the source language and a distinction between numbers treated by themselves and as elements of S-expressions. Some recent versions of LISP allow distinguishing types, but at the time, this seemed incompatible with other features.

d. Free variables. In all innocence, James R. Slagle programmed the following LISP function definition and complained when it didn't work right:

$$\begin{aligned} testr[x,p,f,u] \leftarrow \quad & \mathbf{if}\ p[x]\ \mathbf{then}\ f[x] \\ & \mathbf{else\,if}\ atom[x]\ \mathbf{then}\ u[] \\ & \mathbf{else}\ testr[cdr[x],p,f,\lambda : testr[car[x],p,f,u]]. \end{aligned}$$

The object of the function is to find a subexpression of x satisfying $p[x]$ and return $f[x]$. If the search is unsuccessful, then the continuation function $u[]$ of no arguments is to be computed and its value returned. The difficulty was that when an inner recursion occurred, the value of $car[x]$ wanted was the outer value, but the inner value was actually used. In modern terminology, lexical scoping was wanted, and dynamic scoping was obtained.

I must confess that I regarded this difficulty as just a bug and expressed confidence that Steve Russell would soon fix it. He did fix it but by inventing the so-called FUNARG device that took the lexical environment along with the functional argument. Similar difficulties later showed up in Algol 60, and Russell's turned out to be one of the more comprehensive solutions to the problem. While it worked well in the interpreter, comprehensiveness and speed seem to be opposed in compiled code, and this led to a succession of compromises. Unfortunately, time did not permit writing an appendix giving the history of the problem, and the interested reader is referred to (Moses 1970) as a place to start. (David Park tells me that Patrick Fischer also had a hand in developing the FUNARG device).

e. The "program feature". Besides composition of functions and conditional expressions, LISP also allows sequential programs written with assignment statements and **go to**s. Compared to the mathematically elegant recursive function definition features, the “program feature” looks like a hasty afterthought. This is not quite correct; the idea of having sequential programs in LISP antedates that of having recursive function definition. However, the notation LISP uses for PROGs was definitely an afterthought and is far from optimal.

f. Once the *eval* interpreter was programmed, it became available to the programmer, and it was especially easy to use because it interprets LISP programs expressed as LISP data. In particular, *eval* made possible FEXPRS and FSUBRS which are "functions" that are not given their actual arguments but are given the expressions that evaluate to the arguments and must call *eval* themselves when they want the expressions evaluated. The main application of this facility is to functions that don’t always evaluate all of their arguments; they evaluate some of them first, and then decide which others to evaluate. This facility resembles Algol’s *call-by-name* but is more flexible, because *eval* is explicitly available. A first order logic treatment of “extensional” FEXPRs and FSUBRs now seems possible.

g. Since LISP works with lists, it was also convenient to provide for functions with variable numbers of arguments by supplying them with a list of arguments rather than the separate arguments.

Unfortunately, none of the above features has been given a comprehensive and clear mathematical semantics in connection with LISP or any other programming language. The best attempt in connection with LISP is Michael Gordon’s (1973), but it is too complicated.

h. The first attempt at a compiler was made by Robert Brayton, but was unsuccessful. The first successful LISP compiler was programmed by Timothy Hart and Michael Levin. It was written in LISP and was claimed to be the first compiler written in the language to be compiled.

Many people participated in the initial development of LISP, and I haven’t been able to remember all their contributions and must settle, at this writing, for a list of names. I can remember Paul Abrahams, Robert Brayton, Daniel Edwards, Patrick Fischer, Phyllis Fox, Saul Goldberg, Timothy Hart, Louis Hodes, Michael Levin, David Luckham, Klim Maling, Marvin Minsky, David Park, Nathaniel Rochester of IBM, and Steve Russell.

Besides work on the LISP system, many applications were programmed, and this experience affected the system itself. The main applications that I can remember are a program by Rochester and Goldberg on symbolic computation of impedances and other functions associated with electrical networks, J.R. Slagle’s thesis work on symbolic integration directed by Minsky, and Paul Abrahams' thesis on proof-checking.

5 Beyond LISP 1.5

As a programming language LISP had many limitations. Some of the most evident in the early 1960s were ultra-slow numerical computation, inability to represent objects by blocks of registers and garbage collect the blocks, and lack of a good system for input-output of symbolic expressions in conventional notations. All these problems and others were to be fixed in LISP 2. In the meantime, we had to settle for LISP 1.5 developed at M.I.T. which corrected only the most glaring deficiencies.

The LISP 2 project was a collaboration of Systems Development Corporation and Information International Inc., and was initially planned for the Q32 computer, which was built by IBM for military purposes and which had a 48 bit word and 18 bit addresses, i.e., it was better than the IBM 7090 for an ambitious project. Unfortunately, the Q32 at SDC was never equipped with more than 48K words of this memory. When it became clear that the Q32 had too little memory, it was decided to develop the language for the IBM 360/67 and the Digital Equipment PDP-6 - SDC was acquiring the former, while III and M.I.T. and Stanford preferred the latter. The project proved more expensive than expected, the collaboration proved more difficult than expected, and so LISP 2 was dropped. From a 1970s point of view, this was regrettable, because much more money has since been spent to develop LISPs with fewer features. However, it was not then known that the dominant machine for AI research would be the PDP-10, a successor of the PDP-6. A part of the AI community, e.g. BBN and SRI made what proved to be an architectural digression in doing AI work on the SDS 940 computer.

The existence of an interpreter and the absence of declarations makes it particularly natural to use LISP in a time-sharing environment. It is convenient to define functions, test them, and re-edit them without ever leaving the LISP interpreter. A demonstration of LISP in a prototype time-sharing environment on the IBM 704 was made in 1960 (or 1961). (See Appendix 2). L. Peter Deutsch implemented the first interactive LISP on the PDP-1 computer in 1963, but the PDP-1 had too small a memory for serious symbolic computation.

The most important implementations of LISP proved to be those for the PDP-6 computer and its successor the PDP-10 made by the Digital Equipment Corporation of Maynard, Massachusetts. In fact, the half word instructions and the stack instructions of these machines were developed with LISP's requirements in mind. The early development of LISP at M.I.T. for this line of machines and its subsequent development of INTERLISP (nee BBN LISP) and MACLISP also contributed to making these machines the machines of choice for artificial intelligence research. The IBM 704 LISP was extended to the IBM 7090 and later led to LISPs for the IBM 360 and 370.

The earliest publications on LISP were in the Quarterly Progress Reports of the M.I.T. Research Laboratory of Electronics. (McCarthy 1960) was the first journal publication. The Phyllis Fox was published by the Research Laboratory of Electronics in 1960 and the *LISP 1.5 Programmer's Manual* by McCarthy, Levin, et. al. in 1962 was published by M.I.T. Press. After the publication of (McCarthy and Levin 1962), many LISP implementations were made for numerous computers. However, in contrast to the situation with most widely used programming languages, no organization has ever attempted to propagate LISP, and there has never been an attempt at agreeing on a standardization, although recently A.C. Hearn has developed a "standard LISP" (Marti, Hearn, Griss and Griss 1978) that runs on a number of computers in order to support the REDUCE system for computation with algebraic expressions.

6 Conclusions

LISP is now the second oldest programming language in present widespread use (after FORTRAN and not counting APT, which isn't used for programming *per se*). It owes its longevity to two facts. First, its core occupies some kind of local optimum in the space of programming languages given that static friction discourages purely notational changes. Recursive use of conditional expressions, representation of symbolic information externally by lists and internally by list structure, and representation of program in the same way will probably have a very long life.

Second, LISP still has operational features unmatched by other language that make it a convenient vehicle for higher level systems for symbolic computation and for artificial intelligence. These include its run-time system that give good access to the features of the host machine and its operating system, its list structure internal language that makes it a good target for compiling from yet higher level languages, its compatibility with systems that produce binary or assembly level program, and the availability of its interpreter as a command language for driving other programs. (One can even conjecture that LISP owes its survival specifically to the fact that its programs are lists, which everyone, including me, has regarded as a disadvantage. Proposed replacements for LISP, e.g. POP-2 (Burstall 1968,1971), abandoned this feature in favor of an Algol-like syntax leaving no target language for higher level systems).

LISP will become obsolete when someone makes a more comprehensive language that dominates LISP practically and also gives a clear mathematical semantics to a more comprehensive set of features.

7 References

Abrahams, Paul W. (1963), *Machine verification of mathematical proof,* M.I.T. PhD thesis in mathematics.

Abrahams, Paul W., Barnett, J., et al., (1966), "The LISP 2 Programming Language and System", *Proceedings of the Fall Joint Computer Conference,* pp. 661-676.

Abrahams, Paul W. (1967), *LISP 2 Specifications,* Systems Development Corporation Technical report TM-3417/200/00, Santa Monica, Calif.

Allen, John (1978), *Anatomy of LISP,* McGraw Hill.

Berkeley, Edmund C. and Daniel Bobrow, eds. (1964), *The Programming Language LISP, its Operation and Applications*, Information International Incorporated, Cambridge, Massachusetts. (out of print).

Burstall, R.M., J.S. Collins and R.J. Popplestone (1968), *The POP-2 Papers*, Edinburgh University Press, Edinburgh, Scotland.

Burstall, R.M., J.S. Collins and R.J. Popplestone (1971), *Programming in POP-2.* Edinburgh University Press, Edinburgh, Scotland.

Cartwright, Robert (1976), *A practical formal semantic definition and verification system for typed LISP* , Stanford Artificial Intelligence Laboratory technical report AIM-296, Stanford, California.

Cartwright, Robert and John McCarthy (1978) "Representation of Recursive Programs in First Order Logic" (to be published). (Draft available as FIRST.NEW[W77,JMC] at SU-AI on ARPAnet).

Collins, G.E. (1960) "A method for overlapping and erasure of lists", *Communications of the ACM*, Vol. 3, pp. 655-657.

Church, Alonzo (1941), *Calculi of Lambda conversion,* Princeton University Press, Princeton, New Jersey.

Fox, Phyllis (1960), *LISP I Programmers Manual,* Internal paper, MIT, Cambridge, Mass.

Gordon, Michael (1973) *Models of Pure LISP*, Experimental Programming Reports: No. 31, University of Edinburgh, Edinburgh.

Gelernter, H., J. R. Hansen, and C. L. Gerberich (1960), "A FORTRAN-Compiled List Processing Language", *Journal of the ACM*, Vol. 7, No. 2, pp. 87-101.

Hearn, Anthony (1967), *REDUCE, a User-oriented Interactive System for Algebraic Simplification,* Stanford Artificial Intelligence Laboratory technical report AIM-57, Stanford, California.

Hewitt, Carl (1971), *Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot,* Ph.D. Thesis, MIT, Cambridge, Mass.

McCarthy, John (1958) "Programs with common sense", *Proceedings of the Symposium on the Mechanization of Thought Processes,* National Physiology Lab, Teddington, England.

McCarthy, J., Minsky, M., et al., (1959a), Quarterly Progress Report No. 52, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, J., Minsky, M., et al., (1959b), Quarterly Progress Report No. 55, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, John (1959c), *Letter to the Editor, CACM,* Vol. 2, No. 8.

McCarthy, J., Minsky, M., et al., (1960a), Quarterly Progress Report No. 56, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, John (1960b), "Recursive Functions of Symbolic Expressions and their Computation by Machine, part I",*CACM,* Vol. 3, No. 4, pp. 184-195.

McCarthy, J., Minsky, M., et al., (1962a), Quarterly Progress Report, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, J., Minsky, M., et al., (1962b), Quarterly Progress Report No. 64, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, John (1962c), *LISP 1.5 Programmer's Manual,* (with Abrahams, Edwards, Hart, and Levin), MIT Press, Cambridge, Mass.

McCarthy, J., Minsky, M., et al., (1963a), Quarterly Progress Report No. 68, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, J., Minsky, M., et al., (1963b), Quarterly Progress Report No. 69, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, John (1963c) "A Basis for a Mathematical Theory of Computation", in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, pp. 33-70. North-Holland Publishing Company, Amsterdam.

McCarthy, John (1963d) "Towards a Mathematical Science of Computation", *Proceedings of IFIP Congress, Munich 1962*, Amsterdam: North-Holland, pp. 21-28.

McCarthy, J., Minsky, M., et al., (1965), Quarterly Progress Report No. 76, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, J., Minsky, M., et al., (1966), Quarterly Progress Report No. 80, Research Lab of Electronics, MIT, Cambridge, Mass.

McCarthy, John and Carolyn Talcott (1979) *LISP with Proofs*, to be published. Versions of most chapters are available at the Stanford Artificial Intelligence Laboratory.

Marti, J. B., Hearn, A. C., Griss, M. L.and Griss, C. (1978) *Standard LISP Report,* University of Utah Symbolic Computation Group Report No 60, Provo, Utah.

The Mathlab Group (1977), *MACSYMA Reference Manual,* Laboratory for Computer Science, MIT Version 9, Cambridge, Mass.

Mitchell, R.W. (1964) *LISP 2 Specifications Proposal,* Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif.

Moon, David A. (1974), *MACLISP Reference Manual,* Project MAC Technical Report, MIT, Cambridge, Mass.

Moses, Joel (1970) *The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem*", M.I.T. Artificial Intelligence Memo 199, Cambridge, Mass.

Newell, A., and J.C. Shaw (1957) "Programming the Logic Theory Machine", *Proceedings of the 1957 Western Joint Computer Conference,* IRE.

Rulifson, J. et al. (1968), "QA4 - A Language for Writing Problem-Solving Programs", *Proceeding IFIP 1968 Congress,* TA-2, pp 111-115.

Stoyan, Herbert. Herbert Stoyan of Dresden, DDR has completed several chapters on the history of LISP.

Sussman, G. Winograd, T., and Charniak, E. (1970), *Microplanner Reference Manual,* AI Memo 203, AIL MIT, Camridge, Mass.

Teitelman, Warren (1975), *INTERLISP: Interlisp Reference Manual, Xerox PARC Technical Report, Palo Alto, Calif.*

Weisman, Clark (1967), *LISP 1.5 Primer,* Dickenson Press.

Many reports and memoranda of the M.I.T. and Stanford Artificial Intelligence Laboratories have dealt with various aspects of LISP and higher level systems built on LISP.

APPENDIX - HUMOROUS ANECDOTE

The first on-line demonstration of LISP was also the first of a precursor of time-sharing that we called "time-stealing". The audience comprised the participants in one of M.I.T.'s Industrial Liaison Symposia on whom it was important to make a good impression. A Flexowriter had been connected to the IBM 704 and the operating system modified so that it collected characters from the Flexowriter in a buffer when their presence was signalled by an interrupt. Whenever a carriage return occurred, the line was given to LISP for processing. The demonstration depended on the fact that the memory of the computer had just been increased from 8192 words to 32768 words so that batches could be collected that presumed only a small memory.

The demonstration was also one of the first to use closed circuit TV in order to spare the spectators the museum feet consequent on crowding around a terminal waiting for something to happen. Thus they were on the fourth floor, and I was in the first floor computer room exercising LISP and speaking into a microphone. The problem chosen was to determine whether a first order differential equation of the form $M\,dx + N\,dy$ was exact by testing whether $\partial M/over\partial y = \partial M/over\partial y$, which also involved some primitive algebraic simplification.

Everything was going well, if slowly, when suddenly the Flexowriter began to type (at ten characters per second)

"THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS:"

and on and on and on. The garbage collector was quite new at the time, we were rather proud of it and curious about it, and our normal output was on a line printer, so it printed a full page every time it was called giving how many words were marked and how many were collected and the size of list space, etc. During a previous rehearsal, the garbage collector hadn't been called, but we had not refreshed the LISP core image, so we ran out of free storage during the demonstration.

Nothing had ever been said about a garbage collector, and I could only imagine the reaction of the audience. We were already behind time on a tight schedule, it was clear that typing out the garbage collector message would take all the remaining time allocated to the demonstration, and both the lecturer and the audience were incapacitated by laughter. I think some of them thought we were victims of a practical joker.

John McCarthy Artificial Intelligence Laboratory Computer Science Department Stanford University Stanford, California 94305

This draft of LISP[F77,JMC] PUBbed at time on date.