www-formal.stanford.edu

LISP prehistory - Summer 1956 through Summer 1958.

10–13 minutes

---

next up previous
Next: The implementation of LISP **Up:** History of Lisp **Previous:** Introduction

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 -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.

---

next up previous
Next: The implementation of LISP **Up:** History of Lisp **Previous:** Introduction

John McCarthy
Fri Jul 26 22:37:29 PDT 1996