The implementation of LISP

9–12 minutes

next up previous
Next: From LISP 1 to **Up:** History of Lisp **Previous:** LISP prehistory - Summer

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 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.

---

next up previous
Next: From LISP 1 to **Up:** History of Lisp **Previous:** LISP prehistory - Summer

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