The Influence of the Designer on the Design -- J.~McCarthy and LISP

Herbert Stoyan University of Erlangen D-852 Erlangen, Germany

Abstract

In this chapter, some of the events of LISP development are protocolled. Step by step, the implementers became independent of McCarthy. In 1962 the internal drive was stronger than McCarthy's proposals. The results are somehow ambiguous.

Preparation for the Design

Around 1956 McCarthy understood the central role of a programming language for his scientific goal -- artificial intelligence. A consulting job in 1957 enabled him to experiment with a combination of algebraic notation (as it is used in FORTRAN for describing arithmetical computation) and list processing (as it was invented by Newell, Shaw, and Simon). The experiment was successful and the idea became for him a basis of thinking. He analyzed existing programming languages more deeply (using FORTRAN as model) and started to ask for new means of expression. Before September 1958, McCarthy had proposed in several papers\footnote{A memorandum for the leader of MIT's computer center, his contribution to the US proposal for the new International Algorithmic Language, and a letter to Backus and Turanski.} a list of around 24 new ideas for programming languages. First, he was out to advance the syntactic face of the languages by introducing:

(1) the changeability of typographical conventions,

(2) the possibility to define new terminology (abbreviations),

and

(3) the possibility to describe program variants as substitution operations on programs.

Then he was keen to upgrade the compilers by proposing:

(4) extensibility of programs (incremental compiler) and changeability of programs,

(5) writing the compiler in the language and bootstrapping it,

and

(6) permitting compiler hints in the user program. Next he called for means to:

(7) change the arithmetic and the meaning of operation symbols,

and

(8) generalize the concept of the subscripted variables where the set of subscripts can be any ordered set. He asked for new data types such as:

(9) strings and lists as quantities,

(10) possibilities for manipulating symbolic quantities,

(11) definability of new quantities (algebraic and functional expressions, differential equations, shapes, colors, programs, electrical networks),

(12) functions as quantities,

(13) operations on functions (operations of the range, composition, abstraction, differentiation, integration),

(14) Church's lambda-notation,

(15) higher order functions,

and

(16) multiplet-valued functions\footnote{McCarthy's proposal is not covered by the functionality of CommonLISP for multiple values. In addition to the existing stuff, we need: a macro/specialform {\tt multiple-value-call*} which accepts sequences of functions and applies them to consecutive multiple values, and a macro/specialform {\tt multiple-value-distribute}, which accepts a sequence of multiple values, and rearranges them. (Copying is permitted.) McCarthy proposed a composition expression (written infix), which would lead to one new macro/specialform {\tt multiple-value-compose}.}. Further, he insisted on complete new language constructs:

(17) conditional expressions,

(18) fluents,\footnote{equations acting as demons}

(19) meta propositions (``{\it a} has been done already''),

and,

(20) a direct way of handling propositions and predicates and conditional functions. Finally, he envisioned new ways of programming altogether:

(21) programming by equations between computing states (conditional, collections),

(22) the ability to describe a computation by giving final state of the machine in terms of the initial state without having to worry about intermediate changes to the variables used in the computation,

(23) a way of describing flow apart from the computation statements,

and

(24) the possibility to describe problems by procedures that check solutions.

Not all of these proposals are reality today. Some of them are quite technical; but most of them turned out to be important for programming.

In a proposal for a rigid intermediate language, he pointed out the advantages of Polish notation and of an expression language. We see that McCarthy's thoughts were captured very much by the expressive tool he wanted to realize in order to program intelligent systems and this became possible through the AI project at MIT.

The First Development of LISP

The initial plan {McC58b} for LISP was as an extension of FORTRAN. McCarthy wrote proposals, modeled programs on paper and made his programmers implement the functions by hand-compiling them. There is the unexplained fact that McCarthy, who had already proposed lambda-expressions as part of a programming language (for ALGOL in summer 1958), started into the adventure of designing his algebraic language without taking this idea into account. λ was forgotten the first time.

According to his idea of an intermediate language, he proposed to represent the language by expressions using list structures. It was clear that Polish notation of expressions and the representation of terms as

$f(e_1,...,e_n)$ by {\tt (f, e$_1$,...,e$_n$)}

should be taken. In this way, the external representation of lists and the specification for the read procedure was defined. With his FLPL experience, he invented a series of basic functions for list processing, including {\tt car}, {\tt cdr} (not {\tt cons}!) -- extraction, construction, reference, storage, erase, and pointer moving functions.

The design was based on McCarthy's old ideas, with the exception that arithmetic did not play the central role, as in the algorithmic languages. List processing, algebraic notation, recursive subroutines, conditional expressions, functions as quantities -- these were the new features of the languages.

The following basic quantities (data types) were chosen:

(1) integers -- as addresses,\footnote{McCarthy had not mastered the list as data type: ``While a number of interesting and useful operations on whole lists have been defined, most of the calculations we actually perform cannot as yet be described in terms of these operations. It still seems necessary to compute with the addresses of the elements of the lists."\cite{McC58b}}

(2) whole words,

(3) propositional quantities -- i.e. booleans,

(4) locationals -- program labels,

and

(5) functions. Symbols -- indentified with their p-lists (``each symbol in the program has such a p-list'') -- were not regarded as separate types. The language should contain variables, indexed variables, expressions, conditional expressions and a small set of statements:

(1) a replacement statement -- assignment,

(2) a Go-statement -- transfer of control,

(3) a set-statement -- array-creation,

(4) the subroutine-call,

(5) a compound statement,

(6) iteration statements -- a {\tt do} over an explicitly given list,

(7) declarations -- p-list construction (an imperative act!), and (8) subroutine definition -- a declaration.

Hand-compiling {\tt maplist} convinced McCarthy of a necessary function notation -- which completes the function quantity already proposed. He introduced the lambda-notation after some time, but the proposals for the hand compilation were usable only for compiled function arguments.

Call by value was taken as FORTRAN's strategy.

By reading the old report \cite{McC58c}, one becomes convinced that McCarthy knew enough about the process of hand compilation to enable him to observe the programmers carrying out their work. The programmers were quite clever and did much outside of LISP.\footnote{Model railroads, space war programming, etc.}

The Revision

After two months of paper programming and hand-compiling small functions, J.~McCarthy changed the design nearly to that of a functional language \cite{McC58d}.

The representation of list structures became clearer, symbols and lists\footnote{The empty list is a 15 bit quantity (a pointer) 0, the truth value false a 1 bit quantity 0.} got their structure. It was decided to represent the difference between constants and variables not in notation but as property in the p-lists.

All 3-bit quantities of the first design were dropped. Work with whole words remained only in connection with p-names, {\tt cons} was created from {\tt consel} and {\tt consls}. All storage and pointer moving functions were removed.

Functions were taken as objects with p-lists, i.e., symbols. This was a dangerous step. The intention was to store in the p-list:

the fact that this is a function, a calling sequence, formulas for differentiating and integrating.

The revision was initiated by the decisions for storage structures and the paper programs written so far by McCarthy.

Further Concretization

The next step was the design of storage management for recursive routines. Some FORTRAN standards were taken. McCarthy understood that recursive functions are quite concise but not always efficient. He presented an iterative {\tt maplist} which worked with a local variable {\tt maplist}. It was an imperative version using a cycle. He discussed the ratio of speed penalty versus ease of statement. The language still permitted changing of lists by writing {\tt car} or similar on the left side of an assignment statement.\footnote{Modern version: {\tt (setf (car list) new-car)}.}

In Memo 4 \cite{McC58e}, we find the first usage of the name LISP. Additionally, it contains SAP programs for {\tt cons}, {\tt copy}, {\tt maplist}, {\tt diff}, proving that McCarthy was leading the implementation work.

Substitutional functions were the most advanced things at the end of October. ``The value of a substitutional function applied to a list of arguments is the result of substitutions (substituting) these arguments for the objects on an ordered list of arguments in a certain expression containing these arguments." If we apply the proposed representation to square, we get: {\tt(subfun (x) (times x x))}. λ is not used! This is quite surprising here -- the second time that this happened. The first {\tt apply} is represented as a substituting function:

apply(L,f) = (car(f) = subfun -> sublis(pair(car(cdr(f)),L),car(cdr(cdr(f))))

1 -> error) This {\tt apply} can deliver only list representations of body forms of the function. It is interesting to see the first hint for the necessity of automatic storage management -- ``{\tt apply} is a space thief unless the compiler arranges for the list to be erased once it was used."

The Gap

Memo 4 was written between the end of October and early November. The next memo (7) by McCarthy, was written in spring 1959 \cite{McC59a}. It contains proposals initiating the work on the compiler. In the text we find the first usage of the symbol {\tt cond} as a compiler case.

A Contribution -- Nat Rochester

It seems that during McCarthy's leave Rochester presented his Memo 5 containing the first proposal for an algebraic simplifier \cite{R58}. Between 28th of October and 20th of November, he developed various functions and proposed some alternatives e.g. the compositions of {\tt car} and {\tt cdr}, such as {\tt caadaddr}, the first {\tt append}, {\tt eq}, (full-word) {\tt replace}, a complete {\tt simplify}, an incomprehensible complicated {\tt diff}. Rochester always wrote ``list processing language", instead of LISP.

In a handwritten Memo 5, McCarthy tried to redo Rochester's functions. The script contains only three functions: {\tt conc} = {\tt insert}, {\tt subfp}, {\tt adj} = {\tt var}. He did not use the {\tt car}-{\tt cdr} compositions and proposed (not in the draft) a function {\tt desc}, which accepted a list of {\tt a} and {\tt d} to simulate the work of the composition. In \cite{McC59d}, McCarthy converts to the short {\tt car}-{\tt cdr} composition.

It is hard to comment on this: possibly a tendency to do everything alone? Rochester's {\tt label} proposal was accepted -- in another form.

Two papers for users were written now which meant there were some -- outside the circle of McCarthy, Russell, and Maling. Memo 6 by S.~Russell is on writing and debugging programs. The earlier handwritten memo by McCarthy, ``Notes on Debugging LISP Subroutines," proves again his machine knowledge. It was now possible to input lists. McCarthy had some hope to ``relax input conventions."\footnote{Poor LISPers! This still has not happened!} Assignment is described as: {\tt equal(y, (plus, a, (times, b , c)))} for {\tt y = a+b*c}!

McCarthy started to define the restricted language that is ``the official form of the notation," but did not finish this.

The December Quarterly Progress Report \cite{McC58f} announced a forthcoming RLE tech report describing the LISP system. A small hint in this direction is given: ``{\tt apply} is programmed but not yet debugged. It will be the basis of an interpreter.'' Is that the small {\tt apply} of November, a substituting program, or an a-list using one? It remains unknown.

The First Known Interpreter

Some time between November 1958 and March 1959, McCarthy got the idea to prove the equivalence of his LISP to Turing machines by developing a universal LISP function that simulates LISP functions by processing a list representation of them. As McCarthy reports in \cite{McC78}, S.~Russell took the function, intended for theoretical purposes only, and hand-compiled it. The story is nice, but the implementation purpose is quite obvious. The new functions fulfill the purpose of the {\tt apply} of Memo 4.

In \cite{S84}, I published an {\tt apply}/{\tt eval} as the first known interpreter. It is contained in Memo 8,\footnote{See section 9.} \cite{McC59b}, which is dated the 4th March. In the meantime, we have found an {\tt apply}/{\tt eval} -- which is one day older\cite{MetC59}! In addition, it comes in the tenth modification and is the following program\footnote{To present it as near to the original as possible, I have changed λ to \&, exchanged the handwritten arrows to {\tt ->}, and one $\vee$ to {\tt v}.} (titled ``LISP program for single statement interpreter''):

```
APPLY(F,L,A)=select(car(F);
          -1,app2(F,L,A);
          lambda,eval(caddr(F),append(pair(cadr(F),L),A));
          label,apply(caddr(F),L,append(pair(cadr(F),caddr
                              (F)),A));
          apply(eval(F,A),L,A))
EVAL(E,A)=select(car(E);
          -1,search(A,&(J,caar(J)=E),&(J,cadar(J)),error);
          intv,search(cadr(E),&(J,car(J)=int),&(J,cdadr(J)),
                              error);
          sub,sublis(A,eval(cadr(E),A));
          const,cadr(E);
          label,eval(caddr(E),append(pair(cadr(E),caddr(E)),
                              A));
          varc,search(A,&(J,cadar(J)=cadr(E)),&(J,cadar(J)),
                              error);
          care,search(A,&(J,caar(J)=cadr(E)),&(J,eval(cadar(J),
                              cdr(J)),error);
          apply(car(E),maplist(cdr(E),&(J,eval(car(J),A))),A))
APP2(F,L,A)=select(F;caar,caar(L);cdr,cdar(L);cons,cons(car(L),cadr(L));
          list,L;null,car(L)=0;atom,caar(L)=-1;
                              search(F,&(J,car(J)=subr v expr),
                                   &(J,(car(J)=subr -> app3(F,L,
                                   1 -> apply(cadr(J),L,A))),
          search(A,&(J,caar(J)=F),&(J,apply(cadar(J),L,A)),
                                   error))
evcon(E,A) = (E=0 -> error,eval(caar(E),A) -> eval(cadar(E),A),1 -> evcon
                                     (cdr(E),A))
```

This program is erroneous, obviously. Parentheses are lacking (lines 18, 26) and the end of a line is lost, (line 23, {\tt A);}).

The system is a call-by-value system using a-lists. The a-list is not a list of pairs but a two-element list.

The clauses in {\tt EVAL} correspond to: atom symbols, integers,\footnote{The evaulator delivers the digit representation!} substitutional functions,\footnote{Relic of November 1958.} {\tt quote},\footnote{Named {\tt const}} {\tt label},\footnote{Evaluates the body after associating name and body.} and two cases of variables.\footnote{The second are variables with unevaluated values on the a-list, the first are variables with evaluated values on the a-list -- { \tt cadar} is a mistake and should be changed into {\tt caar}. Both are coded as lists: {\tt (varc x)} the first, and {\tt (vare x)} the second; the {\tt c} must be a writing error. Are they relics of the earlier interpreter versions?} A clause for λ is missing -- the third time λ was forgotten. An explanation might be the start in substitutional functions.

{\tt APPLY} has clauses for functions named by symbols, by lambda-expressions, and by {\tt label}. Other cases are given to {\tt eval} to repeat the evaluation of the function. {\tt APP2} executes {\tt car}, {\tt cdr}, {\tt cons}, {\tt list}, {\tt null}, and {\tt atom} directly. All other implemented functions are activated via their functional property: {\tt subr} and {\tt expr}

are checked together. Afterwards the a-list is checked for local bindings of the symbols. The global definition has preference.

The code looks good\footnote{If we assume hand-compiled {\tt search}. The 4th parameter is an expression that should not be evaluated before executing {\tt search}. Two {\tt pair}-forms are intended for different purposes.} -- with the exception of processing functional arguments, of course. We find this in accordance with it with the comments in the assembly listing of May 1959 (see \cite{S84}). Only the {\tt cond}-clause was added.

Theoretical Interpreters

With a defined interpreter, the development of the language was more for debugging and convenience than for design and deep ideas. Russell did his duty and debugged; McCarthy worked on the theory of S-expressions.

McCarthy has published many interpreters. In \cite{S84}, I have republished versions which were written in March and April 1959. The latter developed into the version published in the CACM-paper \cite{McC60}. All of them are to be regarded as theoretical programs that are on paper only and have no impact on practice.

The first known version \cite{McC59b} of the CACM-paper is dated March 4. The aim of this paper was to prove that LISP is as powerful as Turing machines. For this purpose, McCarthy (1) implemented Turing machines and (2) proved that there is a universal LISP-function.

The important breakthrough was the introduction of symbolic expressions as data type for list structures and the conversion of the basic function from pseudo functions on integers to functions on symbolic expressions.

During this research, McCarthy felt uneasy with the machine dependent names of the selector functions (additionally, {\tt cons} was not regarded as good enough) and he tried to change\footnote{{\tt first} instead of {\tt car}, {\tt rest} instead of {\tt cdr}, {\tt combine} instead of {\tt cons}.} that. Programmers and students were surprised by McCarthy's new language, but nobody took up his proposal and after some days he dropped it. The LISP community was already more powerful as the designer.

For representing LISP-functions as lists, McCarthy had designed the well known translation rules from LISP\footnote{The ``F-expressions", later ``M-Language".} to symbolic expressions.\footnote{ ``S-expressions" became the ``S-language" and then plain LISP.} There was a rule for each part of the language, the constants, the variables and function names, the forms, the null S-expression, the truth values, the conditional expression, the lambda-expressions, the label-expressions, and equalities. These translation rules alone created a lot of trouble later: the introduction of the symbol {\tt NIL} to represent the empty s-expression (= the empty list).\footnote{On March 20 empty lists and sublists still were printed as blanks! Until summer, the empty list was coded as {\tt (intv 0)}. At what stage the interpreter accepted {\tt NIL} for it and {\tt T} for ``true" is not known. The best decision would have been {\tt()}.} In the case of the truth values the f-expressions contained numbers, their translation were unquoted symbols.

```
apply[f;args] = eval[combine[f;args]]
eval[e] = [
first[e]=NULL -> [null[eval[first[rest[e]]]] -> T;1 -> F]
first[e]=ATOM -> [atom[eval[first[rest[e]]]] -> T;1 -> F]
first[e]=EQ -> eval[first[rest[e]]]=eval[first[rest[rest[e]]]] -> T;
     1 -> F]
first[e]=QUOTE -> first[rest[e]];
first[e]=FIRST -> first[eval[first[rest[e]]]];
first[e]=REST -> rest[eval[first[rest[e]]]];
first[e]=COMBINE -> combine[eval[first[rest[e]]];eval[first[rest[rest
     [e]]]]];
first[e]=COND -> evcon[rest[e]];
first[first[e]]=LAMBDA -> evlam[first[rest[first[e]]];first[rest[rest
     [first[e]]]];rest[e]];
first[first[e]]=LABEL -> eval[combine[subst[first[e];first[rest
     [first[e]]];first[rest[rest[first[e]]]]];rest[e]]]]

evlam[vars;exp;args]=[null[vars] -> eval[exp];1 -> evlam[
     rest[vars];subst[first[args];first[vars];exp];rest[args]]]
```

The universal S-function {\tt eval}\footnote{{\tt apply} was based on {\tt eval} completely.} given in the same paper \cite{McC59b} (see above) is a complete interpreter of LISP -- a substituting call-by-name realization of an applied lambda-

calculus. McCarthy did not know this. He realized LISP -- and LISP used lambda-{\em notation}. This may explain why he forgot a clause for lambda-expression in this {\tt eval}. It was the fourth time he forgot λ. There are two main errors: McCarthy ignored the computation of functions and he used a too general substitution procedure. He corrected the latter a little but it remained incorrect because of forgotten variable lists in lambda-expressions.

This leads to the question of McCarthy's knowledge of the lambda-calculus. McCarthy always pointed out that he did not know much about this. It is obvious that he has read the first pages of Church's paper \cite{C41}. The {\tt eval} of March 1958 proves that he did know about substitution. However, there is no sign of renaming variables, no sign of normalization. One cannot believe that McCarthy did not study carefully the substitution procedures of the lambda-calculus for his realization, but this is the only explanation for a series of errors that were not apparent in the simple examples of 1958. Today, we all know much better.

The second theoretical interpreter was published in \cite{McC59c}. (See below.) This changed the substitution procedure into a frozen one -- the a-list was used as in the running system. The interpreter works with a mixture of evaluated and unevaluated arguments. Top level is {\tt apply}. The arguments are quoted and {\tt eval} is activated. Argument subterms are not evaluated if the function is written as lambda-expression. Basic functions have to evaluate their arguments. If there is an expression (or a variable as function name) in functional position, arguments are evaluated. After argument evaluation, the values are quoted. This way, arguments do not get evaluated twice -- but {\tt evlis} strips quotes off and puts them on again: a waste of storage. In the a-list, quoted values and unevaluated arguments are mixed. On access, the values get evaluated in the actual environment, which might be quite different than the original one.

```
apply[f;args] = eval[cons[f;appq[args]];NIL]

appq[m] = [null[m] -> NIL;T -> cons[list[QUOTE;car[m]];appq[cdr[m]]]]

eval[e;a] = [
  atom[e] -> eval[assoc[e;a];a];
  atom[car[e]] -> [
  car[e]=QUOTE -> cadr[e];
  car[e]=ATOM -> atom[eval[cadr[e];a]];
  car[e]=EQ -> [eval[cadr[e];a]=eval[caddr[e];a]];
  car[e]=COND -> evcon[cdr[e];a];
  car[e]=CAR -> car[eval[cadr[e];a]];
  car[e]=CDR -> cdr[eval[cadr[e];a]];
  car[e]=CONS -> cons[eval[cadr[e];a];eval[caddr[e];a]];
  T -> eval[cons[assoc[car[e];a];evlis[cdr[e];a]];a]];
  caar[e]=LABEL -> eval[cons[caddar[e];cdr[e]];cons[list[cadar[e];car[e];a]];
  caar[e]=LAMBDA -> eval[caddar[e];append[pair[cadar[e];cdr[e]];a]]]

evlis[m;p] = [null[m] -> NIL;T -> cons[list[QUOTE;eval[car[m];a]];
             evlis[cdr[m];a]]
```

A functional argument is processed correctly, if a lambda-expression or symbol is written as argument of a form with lambda-expression as function and if the naming variable is directly used in functional position. The other cases are erroneous because a case for λ in {\tt eval} is lacking. A later version \cite{MetC60b} cleared up the differences in argument evaluation -- now strictly call-by-value:

```
apply[f;args] = eval[cons[f;appq[args]];NIL]

appq[m] = [null[m] -> NIL;T -> cons[list[QUOTE;car[m]];appq[
          cdr[m]]]]

eval[e;p] = [
  atom[e] -> [assoc[e;p];
  atom[car[e]] -> [
    eq[car[e];QUOTE] -> cadr[e];
    eq[car[e];ATOM] -> atom[eval[cadr[e];p]];
    eq[car[e];EQ] -> eq[eval[cadr[e];p];eval[caddr[e];p]];
    eq[car[e];COND] -> evcon[cdr[e];p];
    eq[car[e];CAR] -> car[eval[cadr[e];p]];
    eq[car[e];CDR] -> cdr[eval[cadr[e];p]];
    eq[car[e];CONS] -> cons[eval[cadr[e];p];eval[caddr[e];p]];
       T -> eval[cons[assoc[car[e];p];evlis[cdr[e];p]];p]];
    eq[caar[e];LABEL] -> eval[cons[caddar[e];cdr[e]];cdr[e]];
       cons[list[cadar[e];car[e]];p]];
    eq[caar[e];LAMBDA] -> eval[caddar[e];append[pair[cadar[e];
```

```
       evlis[cdr[e];p];p]]]]

evlis[m;p] = [null[m] -> NIL;T -> cons[list[eval[car[m];p];
             evlis[cdr[m];p]]]]
```

The additional {\tt list} in {\tt evlis} is superfluous. Functional arguments are always processed erroneously -- they are forgotten.

Comparing the theoretical interpreters with the realized one of March 1959, we get the feeling of a theory behind practice. The picture of a McCarthy who sits in his room alone and designs a nice theory for purposes of pure science is quite wrong. There is enough written proof that he initiated writing the manual and serving users better. McCarthy worked out the idea of garbage collection and he introduced the ``program feature.'' On the other hand, it was N.~Rochester who organized the first big LISP application -- symbolical computation of electrical networks. Therefore, we shall assume a scenario in which McCarthy became increasingly interested in his theoretical work, delivering still the key ideas for the language and the system, but not greatly controlling the work of the programmers.

Combining Expressions and Statements

The programmers could not program in a function-oriented style. Even McCarthy had discussed a version of {\tt maplist} that was programmed iteratively to save time. In 1958 LISP permitted sequential programming and usage of {\tt go}. The differentiation program designed by McCarthy and maintained by Maling \cite{M59} already contained {\tt return}-forms. Later, {\tt apply} somehow could handle sequential programs. McCarthy refers to this in \cite{McC59d} but does not go into details. We know from recollections of students and programmers that they misused the conditional expression for constructing sequences of statements.\footnote{The function {\tt enterm} in Goldberg's system \cite{G59} is a nice example.} Goldberg shows the {\tt prog}-notation in his thesis but when this was introduced is not known.

It is interesting to note that McCarthy regarded his function-oriented programs as sets of recursive functions, and the statement-oriented programs as real programs.

McCarthy's proposal of May '59 was to write sequences of assignments as a list of pairs of the variable and the expression. Obviously, his ideas of 1958 to code a program by the results of variables were applied. Contrary to the old proposal he now used the list of variables and expressions to represent a sequence of assignments and not a system of equations. An idea to represent the latter is also contained in the paper \cite{McC59d}.

Garbage Collection

Until March '59, the erasure functions played an important role -- at least in the manuals. There is no known LISP code that ever used {\tt eralis} and {\tt erase}. The students who implemented the electrical network system must have written increasingly bigger programs -- and one day the storage was used up. McCarthy often remarked ``this function is a space thief,'' and this was valid for the {\tt apply}/{\tt eval}-interpreter in any case. Now a good idea was necessary. McCarthy delivered the idea of garbage collection. The version of the CACM-paper \cite{McC59c} that appeared in April 1959, in the Quarterly Progress Report, contains the first explanation. The garbage collector was implemented during summer 1959 by S.~Russell and D.~Edwards.

The Funarg Problem

In May already -- if not earlier -- it turned out that the {\tt lambda}-case was missing in {\tt eval}. Functional arguments often occurred as {\tt maplist}, {\tt search}, and other functions did need some, and a named function did not always seem appropriate. Adding quotations was introduced, but this seemed a little strange because labeled lambda-expressions need no quotations. A bug was not assumed -- which suggests that the programmers did not study McCarthy's papers carefully.

In the fall of 1959 J.~Slagle was programming for his integration system SAINT\cite{JS61}. He was probably the first who detected the bug in the way functional arguments were processed. Four years later, Saunders still used Slagle's case as an explanation example. As is well-known, the error shows up as a wrong value of a (relatively) global variable in a function written as lambda-expression and quoted for being accepted as a functional argument. If the function is activated outside its orignal environment, the binding of the global variable has changed. McCarthy admits that he underestimated the problem. ``I must confess that I regarded this difficulty as just a bug and expressed confidence that Steve Russell would soon fix it...'' \cite{McC78}. The analysis proved that the environment for the global variables had to be taken. The solution was the {\tt

funarg} or, better, the closure. The solution was realized as a patch that did not remove the source of the problem -- the missing case for λ in {\tt eval}. Instead, a variant of {\tt quote} was invented, {\tt function}. The first known (practical) interpreter including the {\tt funarg} device is published in \cite{MetC60a}:

```
apply[f;x;a] = [atom[f] -> app2[f;x;a];
     car[f] = LAMBDA -> eval[caddr[f];append[pair[cadr[f];x];a]];
     car[f] = LABEL -> apply[caddr[f];x;append[pairl[cadr[f];
                caddr[f]];a]];
     car[f] = FUNARG -> apply[cadr[f];x;caddr[f]];
                T -> apply[eval[f;a];x;a]]

eval[e;b] = [atom[e]->search[e;&[[j];car[j] = APVAL v car[j] = APVAL1];
                &[[j];caadr[j]];
                &[[j];search[b,&[[j];caar[j] = e];
                          &[[j];cadar[j]];
                          &[[j];error]]]];
     atom[car[e]] ->
       search[cdar[e];&[[j];car[j] = FSUBR v car[j] = SUBR v car[j] =
       FEXPR v car[j] = EXPR];
          &[[j];select[car[j];
              [FSUBR;app3[cadr[j];list[cdr[e];b]]];
              [SUBR;app3[cadr[j];evlis[cdr[e];b]]];
              [FEXPR;apply[cadr[j];list[cdr[e];b];b]];
              apply[cadr[j];evlis[cdr[e];b];b]]]
          &[[j];search[b;
                 &[[j];caar[j] = car[e]];
                 &[[j];apply[cadar[j];evlis[cdr[e];b];b]];
                 &[[j];error]]]];
      T -> apply[car[e];evlis[cdr[e];b];b]]

app2[f;x;a] = select[f;[CAR;caar[x]];[CDR;cdar[x]];
                [CONS;cons[car[x];cadr[x]]];
                [LIST;x];
                search[f;&[[j];[car[j] = SUBR v car[j] = FSUBR]];
                   &[[j];[car[j] = SUBR -> app3[cadr[j];x];
                   T -> apply[cadr[j];x;a]];
                   &[[j];apply[car[sassoc[f;a;error]];x;a]]]]
```

This code would be fine (with exception of the disparity between handling of global functional and normal values), if {\tt lambda} were a {\tt FEXPR} creating the {\tt FUNARG}. The multitude of ``functional'' objects is introduced.

The funarg event proves that McCarthy was now far from his LISP. He won the theoretical battle und searched for the next field. The mathematical theory of computation, time sharing, and proof checking were his new interests. LISP lost first priority.

The Last Blow

The theoretical interpreter of July 1961 has a different face, but there are no functional changes (still without closures) \cite{MetC61}:

```
evalquote[fn;x] = apply[fn;x;NIL]
apply[fn;x;a] = [eq[fn;NIL] -> NIL;
     atom[fn] -> [eq[fn;CAR] -> caar[x];
                  eq[fn;CDR] -> cdar[x];
                  eq[fn;CONS] -> cons[car[x];cadr[x]];
                  eq[fn;ATOM] -> atom[car[x]];
                  eq[fn;EQ] -> eq[car[x];cadr[x]];
            T -> apply[eval[fn;a];x;a]];
     eq[car[fn];LAMBDA] -> eval[caddr[fn];pairlis[cadr[fn];x;a]];
     eq[car[fn];LABEL] -> apply[caddr[fn];cons[con[cadr[fn];
                                       caddr[fn]];a]]];

eval[e;a] = [atom[e] -> cdr[assoc[e;a]];
     [atom[car[e[ -> [
             [eq[car[e];QUOTE] -> cadr[e];
             [eq[car[e];COND] -> evcon[cdr[e];a];
```

```
            T -> apply[car[e];evlis[cdr[e];a];a]]
     T -> apply[car[e];evlis[cdr[e];a];]]

evlis[m;a] = [cons[eval[car[m];a];evlis[cdr[m];a]]];
```

In 1961 while working at the proofchecker, McCarthy detected unexpected behavior of the implemented LISP~1.5 system. He analyzed the situation \cite{McC62} and came to the unfortunate conclusion that his theoretical {\tt eval} and the real running {\tt eval} both had errors. ``Neither of these behaves in the most desirable way; and there exist S-expressions which will be handled correctly by the theoretical version but not by the system version, and conversely. The chief defect of the system eval lies in its handling of functional arguments; the chief defect of the RFSE eval lies in its ignorance of property lists."

``Property list" should be read as ``global environment." For functions, it was represented in the realized system by symbols and their values associated with {\tt expr}, {\tt fexpr}, {\tt subr}, and {\tt fsubr} indicators on the p-list. For other values, it was represented by symbols and their values associated with {\tt apval} indicators on the p-list. The special thing about the working system\footnote{And still is in CommonLISP!} was its double semantics (value spaces) of variables. In functional positions, the value of a symbol was associated with the functional indicators in the p-list. If there was not such a value, the symbol was taken as an ordinary variable and evaluated -- first as a (global) value associated to the {\tt apval} indicator in the p-list and then as a value in the a-list. In argument position, the global function value never would have been consulted -- even if the programmer intended to deliver a function. If the theoretical {\tt eval} contained a representation for the global environment, this would be a solution for this part of the problem. The a-list together with a variable denoting it or another external notation for the global environment would be enough.

McCarthy correctly saw two solutions for this problem: using a permanent a-list or using one value indicator in the p-list.

There was no important difference between theoretical and implemented {\tt eval} regarding the handling of functional arguments. In fact, both of them ignored functional arguments. Instead, the programmer had to ``underline" functional arguments in the running systems. The {\tt function} prefix was needed for lambda-expressions in argument positions only. For variables (symbols), {\tt function} worked as {\tt quote}. The access to global functional values is done in {\tt apply} via one step of indirection using {\tt eval} to change the local functional variable into the function name argument. If one makes global function names evaluable, then this indirection is unnecessary. The local variable gets the functional value of the function name when the arguments are evaluated. For function names, the quotation would become superfluous. For lambda-expressions, the problem of evaluation remains. To solve it, McCarthy proposed the definition of {\tt lambda} as {\tt fexpr}: {\tt ((lambda(e a) (cons 'lambda e)))}. It is obvious that he did not have in mind the {\tt funarg}-problem, i.e., the environment-problem. Therefore, the solution presented is only a half one. But it is easy to add the other half. We are quite near to SCHEME!

McCarthy developed the following interpreter description\footnote{McCarthy introduced super-parantheses: {\tt (..}, {\tt (.}, {\tt (,} etc.} \cite{McC62}:

```
eval (exp; alist) = (..atom(exp) -> search (.exp;
        & ((j);(eq(car(j); VALUE)));
          cadr;
           & (); assoc(exp; alist.)
t -> prog((fnval);
     fnval = eval (car(exp); alist)
     return((fullword(fnval) v eq(car(fnval);LAMBDA)
      V eq(car(fnval);LABEL) -> app l (fnval;
     maplist(,cdr(exp);& ((j);eval(car(j); alist,); alist);
      t -> app l (car(fnval); list(cdr(exp);alist);alist..)
app l(fn;args;alist) = (.
      fullword (fn) -> app 2 (fn; args);
        eq(car(fn);LAMBDA) -> eval(caddr(fn);append(
        pair(cadr(fn),args);alist));
        eq (car(fn);LABEL) -> app l (caddr(fn); args;
       cons (cons(cadr(fn);caddr(fn)); alist.)
evalquote(fn;args) = app l (eval (fn;nil)args;nil)
```

However, the LISP implementors did not use the new memo as a working direction, possibly because of the ``big bulk" of programs already running -- still the only, and weak, argument of the pastors of LISP-errors today.

Since this intervention, McCarthy has not made any more written contributions to LISP.\footnote{But add his lectures, verbal proposals, work on parallel LISP!} Conversion to lambda-calculus, object-oriented extensions, and standardization

did not interest him. He changed to the theory of computation, to program verification, to proof checking, to nonmonotonic logic, etc.

Conclusion

If we compare McCarthy's proposals, his design, and the current state of LISP, then it is remarkable how much was established so early. LISP would look like 3-LISP (without reflection, maybe) if McCarthy had studied Church's paper more closely. As things stand, he must prefer SCHEME to CommonLISP -- a clear, understandable small diamond, to a messy, incomprehensible clump.

Bibliography

\bibitem{C41} A.Church: The Calculi of lambda-Conversion. Princeton, 1941.

\bibitem{G59} S.Goldberg: Solution of an Electrical Network Using a Digital Computer. Masters Thesis, EE Dept., MIT, Cambridge, MA, 1959.

\bibitem{M59} K.Maling: The LISP Differentiation Demonstration Program. MIT AI Memo 10, Cambridge, MA, 1959.

\bibitem{McC58a} J.McCarthy: Some Proposals for the Volume 2 Language. MIT, Cambridge, MA, June 1958.

\bibitem{McC58b} J.McCarthy: An Algebraic Language for the Manipulation of Symbolic Expressions. MIT AI Memo 1, Cambridge, MA, September 1958.

\bibitem{McC58c} J.McCarthy: A Revised Version of MAPLIST. MIT AI Memo 2, Cambridge, MA, September 1958.

\bibitem{McC58d} J.McCarthy: Revision of the Language. MIT AI Memo 3, Cambridge, MA, October 1958.

\bibitem{McC58e} J.McCarthy: Revisions of the Language. MIT AI Memo 4, Cambridge, MA, October 1958.

\bibitem{McC58f} J.McCarthy: Artificial Intelligence Project, Semi-Annual Report, Cumputation Center, MIT, Cambridge, MA, December 1958.

\bibitem{McC59a} J.McCarthy: Notes on The Compiler. MIT AI Memo 7, Cambridge, MA, January (?) 1959.

\bibitem{McC59b} J.McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. MIT AI Memo 8, Cambridge, MA, March 4, 1959.

\bibitem{McC59c} J.McCarthy: XIII. Artificial Intelligence. In: RLE Quarterly Progress Report No.~53, MIT, Cambridge, MA, April 1959.

\bibitem{McC59d} J.McCarthy: Programs in LISP. MIT AI memo 12, Cambridge, May (?) 1959.

\bibitem{McC60} J.McCarthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine. CACM, Vol. 3 (1960), No.3, p.184-195.

\bibitem{McC62} J.McCarthy: A New Eval Function. MIT AI Memo 34, Cambridge, MA, 1962 (?).

\bibitem{McC78} J.McCarthy: History of LISP. ACM SIGPLAN Notices, Vol. 13 (1978), No. 8, p.217-223.

\bibitem{MetC59} J.McCarthy, S.Russell, K.Maling, N.Rochester, S.Goldberg, J.Slagle: LISP Programmer's Manual. MIT AI Project, Cambridge, MA, March 1959.

\bibitem{MetC60a} J.McCarthy, R.Brayton, D.Edwards, P.Fox, D.Luckham, K.Maling, D.Park, S.Russell: Preliminary LISP Programmer's Manual. Draft. Computation Center and RLE, MIT, Cambridge, MA, January 1960.

\bibitem{MetC60b} J.McCarthy, R.Brayton, D.Edwards, P.Fox, D.Luckham, K.Maling, D.Park, S.Russell: LISP I Programmer's Manual. Computation Center and RLE, MIT, Cambridge, MA, March 1960.

\bibitem{MetC61} J.McCarthy, M.Minsky, P.Abrahams, R.Brayton, D.Edwards, L.Hodes, D.Luckham, M.Levin, D.Park, T.Hart: LISP 1.5 Programmer's Manual. MIT AI Project, Cambridge, MA, July 1961.

\bibitem{R58} N.Rochester: Symbol Manipulation Language. MIT AI Memo 5, Cambridge, MA, November 1958.

\bibitem{JS61} J.Slagle: A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus. Ph.D.Thesis, Dept. of EE, MIT, Cambridge, MA, 1961.

\bibitem{S84} H.Stoyan: Early LISP History (1956-1959). Conf.Rec. 1984 ACM Symposium on LISP and Functional Programming. ACM, 1984.

Two Further Interpreters

A second version in the manual of March 1960:

```
apply[f;x;p] = [atom[f] -> app2[f;x;p];
      car[f] = LAMBDA -> eval[caddr[f];append[pair[cadr[f];
               x];p]];
      car[f] = LABEL -> apply[caddr[f];x;append[list[list[
               cadr[f];caddr[f]]];p]];
      car[f] = FUNARG -> apply[cadr[f];x;caddr[f]];T ->
               apply[eval[f;p];x;p]]

eval[e;b] = [atom[e]->search[e;&[[j];eq[car[j];APVAL] v
            eq[car[j];APVAL1]];caadr;search[b;
            &[[j];eq[caar[j];e]];&[[j];cadar[j]];
            &[[j];error]]]];
 atom[car[e]] ->
   search[cdar[e];&[[j];eq[car[j];FSUBR]v eq[car[j];
   SUBR]v eq[car[j];FEXPR]v eq[car[j];EXPR]];
   &[[j];select[car[j];
   [FSUBR;app3[cadr[j];list[cdr[e];b]]];
   [SUBR;app3[cadr[j];evlis[cdr[e];b]]];
   [FEXPR;apply[cadr[j];list[cdr[e];b];b]];
   apply[cadr[j];evlis[cdr[e];b];b]]];
   &[[j];search[b;
   &[[j];eq[caar[j];car[e]]];
   &[[j];apply[cadar[j];evlis[cdr[e];b];b]];
   &[[j];error]]]];
   T -> apply[car[e];evlis[cdr[e];b];b]]
```

The version published in the CACM-paper, March 1960:

```
apply[f;args] = eval[cons[f;appq[args]];NIL]

appq[m] = [null[m] -> NIL;
       T -> cons[list[QUOTE;car[m]];appq[cdr[m]]]]

eval[e;a] = [
  atom[e] -> assoc[e;a];
  atom[car[e]] -> [
  eq[car[e];QUOTE] -> cadr[e];
  eq[car[e];ATOM] -> atom[eval[cadr[e];a]];
  eq[car[e];EQ] -> [eval[cadr[e];a] = eval[caddr[e];a]];
  eq[car[e];COND] -> evcon[cdr[e];a];
  eq[car[e];CAR] -> car[eval[cadr[e];a]];
  eq[car[e];CDR] -> cdr[eval[cadr[e];a]];
  eq[car[e];CONS] -> cons[eval[cadr[e];a];eval[caddr[e];
          a]]; T -> eval[cons[assoc[car[e];a];
                                evlis[cdr[e];a]];a]];
  eq[caar[e];LABEL] -> eval[cons[caddar[e];cdr[e]];
                       cons[list[cadar[e];car[e]];a]];
  eq[caar[e];LAMBDA] -> eval[caddar[e];
               append[pair[cadar[e];evlis[cdr[e];a];a]]]

evlis[m;p] = [null[m] -> NIL;
           T -> cons[eval[car[m];a];evlis[cdr[m];a]]]
```

Corrected Versions

A corrected version 1 (read ``funarg" for ``closure" if you want): P-Lists are removed and the {\tt lambda}-case is introduced.

```
APPLY(F,L,A)=select(car(F);
          -1,app2(F,L,A);
          lambda,eval(caddr(F),append(pair(cadr(F),L),A));
          closure,eval(caddadr(F),append(pair(cadadr(F),L),
                                         caddr(F)));
          label,apply(caddr(F),L,append(pair(cadr(F),caddr
                           (F)),A));
          apply(eval(F,A),L,A))

EVAL(E,A)=select(car(E);
          -1,search(A,&(J,caar(J)=E),&(J,cadar(J)),error);
          const,cadr(E);
          label,eval(caddr(E),append(pair(cadr(E),caddr(E)),
                           A));
          lambda,list(closure,E,A);
          closure,E;
          apply(car(E),maplist(cdr(E),&(J,eval(car(J),A))),A))

APP2(F,L,A)=select(F;caar,caar(L);cdr,cdar(L);cons,cons(car(L),cadr(L));
          list,L;null,car(L)=0;atom,caar(L)=-1;
          search(A,&(J,caar(J)=F),
                   &(J,(subrp(cadar(J)) -> app3(F,L,A);
                        1 -> apply(cadar(J),L,A))),
                   error))

evcon(E,A) = (E=0 -> error,eval(caar(E),A) -> eval(cadar(E),A),
              1 -> evcon(cdr(E),A))
```

Corrected version of second interpreter: computation of lambda-expressions is included -- a case for lambda-expressions in {\tt eval} is necessary now.

```
apply[f;args] = eval[combine[f;args]]

eval[e] = [
first[e]=NULL -> [null[eval[first[rest[e]]]] -> T;1 -> F]
first[e]=ATOM -> [atom[eval[first[rest[e]]]] -> T;1 -> F]
first[e]=EQ -> eval[first[rest[e]]]=eval[first[rest[rest[e]]]] -> T;
     1 -> F]
first[e]=QUOTE -> first[rest[e]];
first[e]=FIRST -> first[eval[first[rest[e]]]];
first[e]=REST -> rest[eval[first[rest[e]]]];
first[e]=COMBINE -> combine[eval[first[rest[e]]];eval[first[rest[rest
     [e]]]]];
first[e]=COND -> evcon[rest[e]];
first[e]=LAMBDA -> e;
first[first[e]]=LAMBDA -> evlam[first[rest[first[e]]];first[rest[rest
     [first[e]]]];rest[e]];
first[first[e]]=LABEL -> eval[combine[subst[first[e];first[rest
     [first[e]]];first[rest[rest[first[e]]]]];rest[e]]]];
1 -> eval[combine[eval[first[e]],rest[e]]]]

evlam[vars;exp;args]=[null[vars] -> eval[exp];1 -> evlam[
     rest[vars];subst[first[args];first[vars];exp];rest[args]]]
```