Early LISP History (1956 - 1959)

by

Herbert Stoyan
University of Erlangen-Nürnberg
Martensstraße 3, D-8520 Erlangen
W-Germany

ABSTRACT

This paper describes the development of LISP from McCarthy's first research in the topic of programming languages for AI until the stage when the LISP1 implementation had developed into a serious program (May 1959). We show the steps that led to LISP and the various proposals for LISP interpreters (between November 1958 and May 1959). The paper contains some correcting details to our book (32).

INTRODUCTION

LISP is understood as the model of a functional programming language today. There are people who believe that there once was a clean "pure" language design in the functional direction which was compromised by AI-programmers in search of efficiency. This view does not take into account, that around the end of the fifties, nobody, including McCarthy himself, seriously based his programming on the concept of mathematical function. It is quite certain that McCarthy for a long time associated programming with the design of stepwise executed "algorithms".

On the other side, it was McCarthy who, as the first, seemed to have developed the idea of using functional terms (in the form of "function calls" or "subroutine calls") for every partial step of a program. This idea emerged more as a stylistic decision, proved to be sound and became the basis for a proper way of programming - functional programming (or, as I prefer to call it, function-oriented programming).
We should mention here that McCarthy at the same time conceived the idea of logic-oriented programming, that is, the idea of using logical formulae to express goals that a program should try to establish and of using the prover as programming language interpreter.
To come back to functional programming, it is an important fact that McCarthy as mathematician was familiar with some formal mathematical languages but did not have a deep, intimate understanding of all their details. McCarthy himself has stressed this fact (23). His aim was to use the mathematical formalismus as languages and not as calculi. This is the root of the historical fact that he never took the Lambda-Calculus conversion rules as a sound basis for LISP implementation. We have to bear this in mind if we follow now the sequence of events that led to LISP.
It is due to McCarthy's work that functional programming is a usable way of programming today. The main practice of this programming style, done with LISP, still shows his personal mark.

A programming language for artifical intelligence

It seems that McCarthy had a feeling for the importance of a programming language for work in artifical intelligence already before 1955. In any case, the famous proposal for the Darmouth Summer Research Project on Artificial Intelligence - dated with the 31th of August 1955 - contains a research program for McCarthy which is devoted to this question: "During next year and during the Summer Research Project on Artificial Intelligence, I propose to study the relation of language to intelligence ..." (25).

A PROPOSAL FOR THE

DARTMOUTH SUMMER RESEARCH PROJECT

ON ARTIFICIAL INTELLIGENCE

J. McCarthy, Dartmouth College
M. L. Minsky, Harvard University
N. Rochester, I. B. M. Corporation
C. E. Shannon, Bell Telephone Laboratories

August 31, 1955

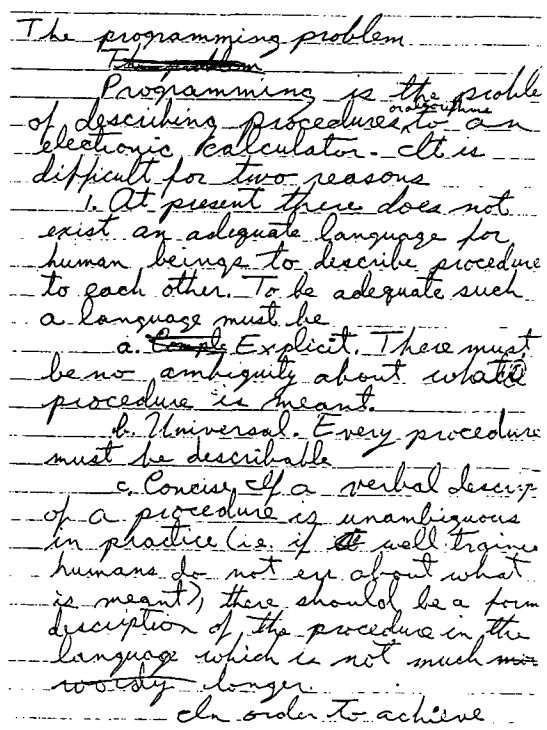
McCarthy compared (formal) logical and natural languages in this proposal for his personal research program. His conclusion was: "It therefore seems to be desirable to attempt to construct an artifical language which a computer can be programmed to use on problems and self-reference. It should correspond to English in the sense that short English statements about the given subject matter should have short correspondents in the language and so should short arguments or conjectural arguments." His goal: "I hope to try to formulate a language having these properties and in addition to contain the notions of physical object, event., etc., with the hope that using this language it will be possible to program a machine to learn to play games well and do other tasks." (25)

It's the natural problem of our kind of historical research that most written outlines, sketches or drafts of papers were destroyed long ago our put into the paper basket during their early existence. We have in our possession a handwritten paper, which seems to be of this time, which helps to shed light on this stage of the development of McCarthy's ideas. It has the title: "The Programming Problem". McCarthy wrote: "Programming is the problem of describing procedures or algorithms to an electronic calculator. It is difficult for two reasons:

1. At present there does not exist ad adequate language for human beings to describe procedures to each other. To be adequate such a language must be
   a. explicit: There must be no ambiguity about what procedure is meant;
   b. universal: Every procedure must be describable;
   c. concise: If a verbal description of a procedure is unabmiguous in practice (i.e. if well trained humans do not err about what is meant), there should be a formal description of the procedure in the language which is not much longer.

   In order to achieve conciseness it seems to be necessary to be able to define new terminology in the language.
   In fact, the language whould be universal in the technical sense that any other formal procedural language can be defined in the universal one. Then the latter can be used to describe procedures for which it is especially suited.

2. The second difficulty is that empty computers have to be instructed in languages fixed at the computer was designed and which are stongly affected by engineering considerations ..." (10)


The programming problem
Programming is the problem of describing procedures to an electronic calculator. It is difficult for two reasons
1. At present there does not exist an adequate language for human beings to describe procedure to each other. To be adequate such a language must be
a. Explicit. There must be no ambiguity about what procedure is meant.
b. Universal. Every procedure must be describable
c. Concise. If a verbal descrip of a procedure is unambiguous in practice (i.e. if well trained humans do not err about what is meant), there should be a form description of the procedure in the language which is not much longer
In order to achieve

We don't know on which level McCarthy's thinking was, when the Summer Project on AI was going on at the Dartmouth College (where McCarthy had an assistant professorship at this time). It is well known (29) that this project was visited by O. Selfridge and R. Solomonoff (both from MIT), A. Bernstein, T. Moore and A. Samuel (all of IBM), J. Bigelow and D. Sayre - and last but not least, by A. Newell and H. Simon. The invitors were J. McCarthy, M. Minsky, N. Rochester and C. Shannon. Newell and Simon could already show results at this time. They had a running program, their papers were accepted for conferences and they gave many talks concerning their work. They had programmed the RAND-Corporation's JOHANNIAC to prove and manipulate logical theorems (sentence calculus). Because of the contemporary overestimation of computer hardware they called their system a "logic theory machine". Today, they would have spoken, even at the beginning of a new programming language. Newell and Simon had an understanding of the importance of expressive means available in such a system. The new thing was that the logical language LL in their "information processing" system was intended to be equipped with facilities for list processing - "one of the most important ideas which ever came up in programming" (35).
We don't want to discuss too many details of the work of Newell and Simon; (29) contains interesting illustrations.

It is important for our topic that McCarthy was not very impressed by this language - which was called IPL. Because of its low level (i.e. near to the machine) "deferred from the format of a loader for the JOHANNIAC which was available to them ..." (23), he did not believe in this language's value for overcoming difficult programming problems. There are good reasons to assume that McCarthy who was

then a "visitor" at IBM did know about the development of IBM's formula translator FORTRAN. In Spring of 1956 the first reports had been published which described its characteristics and power (8) - after the publication of the design in 1954 (7).

It is one of the most important events in the history of programming that McCarthy, who was looking for a mathematical-logical programming language, found interesting elements of it in FORTRAN. He was fascinated by the idea of writing programs with "algebraic" means, i.e. mathematical expressions.

The following two years, McCarthy was working to develop the idea of functional programming - programming with composed terms without an explicit sequence of execution.
Already in 1956 it was clear that one had to work with symbolic expressions to reach the goal of artifical intelligence. As the researchers already understood numerical computation would not have much importance. McCarthy, in his aim to express simple, short description parts by short language elements, saw the composition of algebraic subexpressions as an ideal way to reach this goal.
McCarthy was strongly impressed by the list processing idea of Newell and Simon. List structures then sometimes called "NSS -memory" (2) seemed to be very flexibly and suitable for representing complex logical-symbolic information.
To exploit the advantages of list processing inside a FORTRAN-like language McCarthy decided one need only enlarge the set of standard functions by functions for this purpose. It was a happy coincidence that as early as 1957, not only he could test his idea for reasibility, but show how well it fits the problem.

At the beginning of 1957 N. Rochester (who was at this time head of the Information Science Department of the Poughkeepsie Research Center of IBM) initiated the realization of an idea of M. Minsky that he had developed during the Dartmouth Summer Project:
The plan was to implement a program for proving simple theorems of geometry. H. Gelernter, who had earned his Ph.D. shortly before, was interested in working on this project and McCarthy was in charge, acting as consultant.
Despite proposals to transfer IPL to the IBM704 and use it for the project, McCarthy was successful with his suggestion to enlarge FORTRAN by functions for list processing. "He pointed out that the nesting of functions that is allowed within the FORTRAN format makes possible the construction of elaborate information-processinng subroutines with a single statement." (2) We don't know how detailed McCarthy's proposals were for doing list processing inside FORTRAN. He definitely has always regarded the most important part of these proposals - which proved to be essential for the success of the geometry-prover project - as his own property.

If we look into the paper (2) we will detect many of the functions, with slightly altered names, that later became important in LISP. It seems that in this time any function that delivered integer values had to have a first letter X. Any function (as opposited to subroutines) had to have a last letter F in its name. Therefore the functions selecting parts of the IBM704 memory register (word) were introduced to be XCSRF, XCPRF, XCDRF, XCTRF and XCARF.

S	1 2	3 17	18-20	21-35
1 bit of data	Type code	Address of location word for next entry on this list. Set to zero in terminating *l*-word.	3 bits of data	Address of data (or list) entered on this list, or else 15 bits of data.
Sign	Prefix	Decrement	Tag	Address

Standard *l*-word format

The programmers (and McCarthy) had big difficulties in accepting the functions for construction (XDWORDF and XLWORDF) as proper functions because they unterstood the selection functions to be integer functions, i.e. functions that map integers onto integers. "The dominant characteristic of most of our special list-processing functions is that they are not functions at all in the normally understood sense of the term. For many of them, the value of the function depends not only upon its arguments, but also upon the particular internal state configuration of the computer at the time it is evaluated. Too, some of these functions not only have numerical values, but also produce changes in the internal state configuration of the computer as they are 'evaluated'" (2).

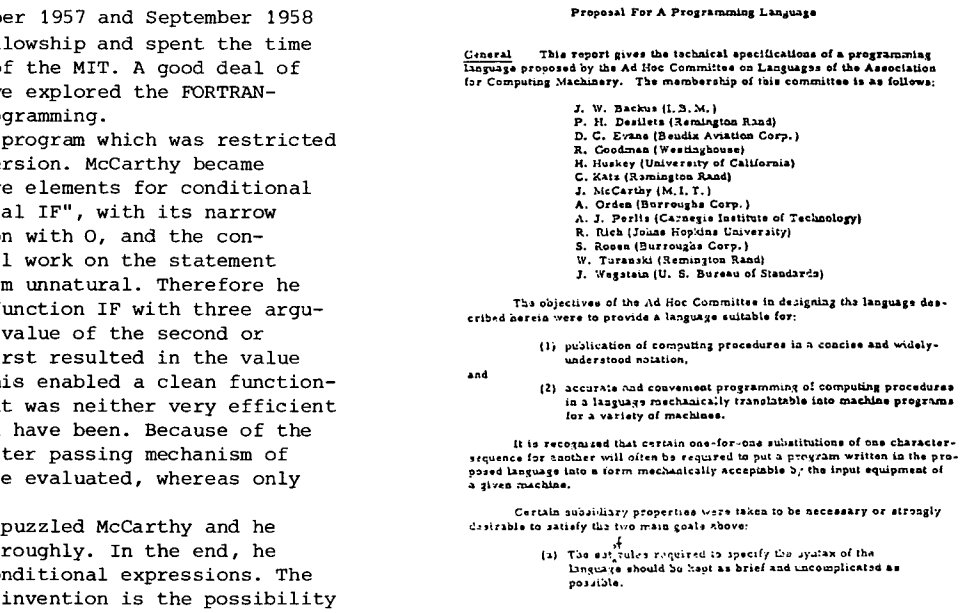
The thinking model suggested by FORTRAN was self-evident in such a degree to all involved, that they couldn't see lists behind the addresses of list structures, but only integers. Therefore McCarthy had doubts about composing terms for list construction. It was Gelernter and Gerberich who ignored these doubts and showed that programming becomes quite elegant if one composes those terms too. Only after two further years McCarthy could leave the world of FORTRAN-concepts and understood that here, a completely new domain is to be assumed, the domain of symbolic expressions, which is represented by address structures.
It should be well known that the geometry-prover project proceeded with much success (3, 4). The set of singular functions added to FORTRAN for list processing evolved into a real system which was then called FLPL (FORTRAN LIST PROCESSING LANGUAGE) (6).
We dont't know very much about McCarthy's work between Summer 1956 and the beginning of 1958, which was related to the functional programming language for list processing, besides the consultation to the geometry-prover project. One paper, (9), contains some speculation on the utility of functions that produce more than one value. The dayly work at Dartmouth College does not seem to have had much lasting influence on his work. We should mention that sometime during 1957 McCarthy met Steve Russell, then a student of mathematics at Dartmouth, who was willing to give up mathematics, which he did not like very much, and to work for McCarthy. However with the exception of a introductory course in assembly programming for the IBM704, not very much seems to have happened during 1957.

We hear that McCarthy gave lectures in a summer school at the University of Michigan which was run by J. Carr III in 1957. He tried there to communicate his idea of functional programming. It seems that he became known as computer scientist who had interesting thoughts concerning programming languages.

The year between September 1957 and September 1958 McCarthy got a Sloan-fellowship and spent the time at the computer center of the MIT. A good deal of the time he seems to have explored the FORTRAN-means for functional programming.
The example was a chess program which was restricted to the normal FORTRAN version. McCarthy became unhappy with the language elements for conditional actions. The "arithmetical IF", with its narrow restriction to comparison with 0, and the constraint to do conditional work on the statement level, only seemed to him unnatural. Therefore he invented a conditional function IF with three arguments. It delivered the value of the second or third argument if the first resulted in the value 1 or 0, respectively. This enabled a clean function-oriented programming, but was neither very efficient nor as nice as it should have been. Because of the call-by-reference parameter passing mechanism of FORTRAN all arguments are evaluated, whereas only two are actually needed.
This unused computation puzzled McCarthy and he explored the problem thoroughly. In the end, he developed the idea of conditional expressions. The special feature of this invention is the possibility of composing conditional expressions and restricting the computation to those really needed to get the value of the whole expression. In contrast to a IF-statement and an IF-function, one obtains an increase in expressive power and efficieny - which leads to completely new applications. We have again no concrete date for this research; all we know is it was completed in April 1958.

A second project, which McCarthy pursued in the fall of 1957, was the extension of the class of possible FORTRAN-functions. FORTRAN I, as it was at this time, offered the facility of defining new functions if the programmer could express them on one line by a term. This facility was a very recent addition, which was not described in the manual. The uninformed user was therefore restricted to standard functions (on the library tape). It is evident that the class of functions definable by using some 60 characters is not very large. Therefore McCarthy developed subprograms which enabled the conversion of FORTRAN-programs into library tape functions. The program for this purpose is described in (26).
At the end of 1957 J. Carr III formed the ACM ad-hoc-committee for programming languages. He asked P. Morse at the MIT computer center to send an interested member, and it seems that he thought of McCarthy. In fact, McCarthy got this delegation. The committee met first on 1/25/58 and, on the meeting on 18th April it 1958 created a subcommittee, which was ordered to develop a ACM-proposal for an international algorithmic language. Members of this subcommittee were J. Backus, J. McCarthy, A. Perlis and W. Turanski.

The proposal was written during April and Mai 1958; its title is "Proposal for a Programming Language". We can identify McCarthy's ideas in all parts where conditional expressions are dealt with or referenced. It was the paper that the ACM delegates had with them when tey set out for Zürich at the beginning of June 1958 to meet the delegation of the GAMM.



Proposal For A Programming Language

General This report gives the technical specifications of a programming language proposed by the Ad Hoc Committee on Languages of the Association for Computing Machinery. The membership of this committee is as follows:

J. W. Backus (I.B.M.)
P. H. Desilets (Remington Rand)
D. C. Evans (Bendix Aviation Corp.)
R. Goodman (Westinghouse)
H. Huskey (University of California)
C. Katz (Remington Rand)
J. McCarthy (M.I.T.)
A. Orden (Burroughs Corp.)
A. J. Perlis (Carnegie Institute of Technology)
R. Rich (Johns Hopkins University)
S. Rosen (Burroughs Corp.)
W. Turanski (Remington Rand)
J. Wegstein (U. S. Bureau of Standards)

The objectives of the Ad Hoc Committee in designing the language described herein were to provide a language suitable for:

(1) publication of computing procedures in a concise and widely-understood notation,

and

(2) accurate and convenient programming of computing procedures in a language mechanically translatable into machine programs for a variety of machines.

It is recognized that certain one-for-one substitutions of one character-sequence for another will often be required to put a program written in the proposed language into a form mechanically acceptable by the input equipment of a given machine.

Certain subsidiary properties were taken to be necessary or strongly desirable to satisfy the two main goals above:

(a) The set of rules required to specify the syntax of the language should be kept as brief and uncomplicated as possible.

If we look into this paper (28), we are surprised by the role played by the conditional expressions. They are not listed as a kind of normal (numerical) expressions. Instead they are one of of the alternatives for expressions that can occur as the argument of a GOTO-statement! These expressions are called "designational expressions". We see here the now well-known notation

$$(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n)$$

Moreover, there is a kind of statement, the conditional statement, which is introduced by a recursive definition, and has in general the same form as the conditional expressions:

$$P_1 \rightarrow S_1, P_2 \rightarrow S_2, \ldots, P_m \rightarrow S_m$$

Where P_i is a Boolean expression and S_j a primitive statement ("module") or a conditional statement.

It seems to be near to the truth to attribute to McCarthy the further "substitution statement", a strange construct, which helps "to avoid rewriting segments of program with minor changes ..." ( ). By

$$s(s'_1 \rightarrow s_1, s'_2 \rightarrow s_2, \ldots, s'_n \rightarrow s_n)$$

where s is the label of a sequence of statements given by a LABEL statement, each s_i is a symbol appearing in the labelled sequence, and the corresponding s'_i is a symbol. The statement is executed by executing the segment s where all s_i are substituted by the corresponding s'_i. In modern terms, we would call the substitution statement a "Macro".

It is known that conditional expressions were not all the proposals that McCarthy had made for the new programming language. Most of them were rejected already by the subcommittee or adjourned and put into a "second volume". In Zürich, the conditional expressions and the conditional statement were also rejected. McCarthy was not a member of the ACM-delegation and therefore his recollection: "I made a lot of propaganda for the inclusion of conditional expressions in ALGOL ... and in 1958 I lost, namely the idea was to unfamiliar, and I didn't explain it well enough, so I did not succed in getting this thing into ALGOL in 1958 ..." (22) does not explain what happened. If we look at the way the conditional

expressions were presented, we might agree with McCarthy that they were badly explained - but we do not believe that the lack of familarity with the new concept was an important reason.
Around the same time, during May 1958, McCarthy took the opportunity to give a lecture in Minsky's course at the departement of mathematics. The topic was 'An Algebraic Coding System". (We do not discuss here two other lectures which McCarthy gave in the same course, one on "advice-taking machines" and one on "chess programs".) The course was given for graduate students and we owe to J. Slagle our information about this event. (It was J. Slagle's fate to become blind at this time and he could not quite make Braille notes. Therefore he has saved only very short fragments of McCarthy's thoughts.)
Slagle wrote: "FORTRAN plus variable functions, composite functions (Church lambda), multiple functions, several valued functions of several variables, direct sum of functions, label portion of program with facility to make symbolic substitutions therein ...".
It is obviously hard to infer very much concerning the state of McCarthy's view of the matter from these fragmentary notes. It seems to be clear that he thought a lot about the possibilities of composing functions. The juxtaposition of "composite functions" and "lambda calculus" in Slagle's telegraphic style is not very informative. The remarks on functions with multiple values repeat the content of (28). The sentence concerning the "Label portion" "to make ... substitutions therein" reminds us of the substitution statement of the ACM programming language proposal. To sum up: we get the strong feeling that the ACM proposal was written entirely by McCarthy and we should note the first reference to the Lambda Calculus made in the context of programming languages. As we will see, this is significant.
At the begin of June, McCarthy again wrote a paper concerning the international algorithmic language. His aim was to rediscuss points that were covered in the meetings preparing the Zürich meeting but discarded. The paper, titled "Some Proposals for the Volume 2 (V2) language", is addressed to Perlis and Turanski. McCarthy tried to clarify his points. He started: "The material that was cut out of volume 1 and not subsequently restored does not amount enough to justify a volume 1 1/2. Therefore I think we should not try to produce an immediate report but should aim after long range goals ...".
The first part of the paper (11) discussed the intermediate language, which was to be convenient for a compiler. McCarthy stressed the advantage of expressing a whole program as a composite expression and proposed the prefix-notation.
The subject of the second part are functions. McCarthy points out that the language described by the first volume contains functions only as constant entities, i.e. a symbol denotes only one and the same function. He proposed including variables for functions and permitting them on the left side of assignment statements. To increase the potential applications of these language elements he proposed additional operations with functions:

"1. Addition, subtraction, multiplication and divison for numerical valued functions. In general we shall want any operations which were appropriate on the range of a set of functions.
2. Composition ... is appropriate whenever the domain of (a function) f and the range of (another function) g coincide.
3. Abstraction from forms. ... I have chosen to propose that we use the Church lambda notation ... This implies that we must also admit forms into our system and an appropriate collection of operations on them ...
4. Operations on functions such as differentiation, other differential operators, and integration ...
5. ...direct sum operation (of multiplet valued functions) ..."

Then he considers permitting the description of logical relations between variables (which could be understood as a consequence of the work on the advice taker which developed into logic-oriented programming).
The end of this interesting paper contains the proposal for using rules for compiler construction: "We envisage the compiler as a program which translates the text according to rules. We hope that most of these rules can be given by formulas ... If this can be done the compiler will be very easily described and it will be very easy for the programmer to introduce new notations." We don't want to stylize these short remarks into the founding of a new rule-oriented programming style. However, we believe this to be an important idea, which for a long time was forgotten. It was this author who proved the feasibility of the idea of a rule-based compiler (33, 34).
The summer of 1958 McCarthy spent at various summer schools and at IBM. At the summer school of the University of Michigan he taught a student Klim Maling, who was looking for a job. McCarthy hired Maling for his and Minskys new Artifical Intelligence Project which was scheduled to be started at September 1st at MIT Cambridge. (The second programmer position had already been given to S. Russell.)
In the rest of the time McCarthy pursued two things: First he wrote his "Advice Taker" paper for the symposium on the mechanization of thought processes in Teddington (England) and he developed the idea of enabling programmers to do operations on functions inside an algorithmic language.
A version of the advice taker paper dated 7/24/58 has survived, which seems to be more detailed than the published version (16). In the section "The construction of the advice taker" McCarthy describes the representaion of logical programs as list structures. The expressions are defined recursively from atomic expressions called terms and sequences of expressions. These expressions are regarded as declarative sentences in a logical system which is understood to be similar to a Post cannonical system with the production rules substitution and modus ponens. It is obvious that McCarthy planned a logic-oriented programming system. Due to the low state of art in automated theorem proving this idea could not become realized in 1958.
The work on the function manipulation operations became contrated on symbolic differentiation. McCarthy who seems not to have seen the importance of recursive functions for symbolic information processing before became aware or this because "the idea of differentiation is obviously recursive" (24). Additionally the functional arguments for functions proved to have their worth. It was the example of differentiating a sum of expressions which makes it necessary to apply the differential operator to every summand, which induced the idea of the map-list function. McCarthy found it quite natural and clear to use a function that applies a given

functional argument to every element of a list, and results in the list of all values. This created the problem of functional arguments.
It was obvious now that all these interesting things could not simply be added to FORTRAN. Conditional expressions, recursion, functional arguments are not simply additions but require a new language. The IAL (ALGOL58) developed in another direction. Therefore McCarthy embarked on the project to develop his own programming language.

The development of LISP. It was McCarthy's fortune that he found a financial basis for his work: The MIT Artifical Intelligence Project was founded on the first of September, 1958. McCarthy became an assistant professor in the departement of Electrical Engineering (his branch was communication sciences), Minsky was already assistant professor in the departement of mathematics. They received support from the Research Laboratory for Electronics in the form of the two programmers, a secretary, a typewriter and six graduate students.

When Maling started his job at the first of September he met S. Russell who seems to have something to do already. Maling remembers: "We were housed in an office in the basement of one of MIT's buildings, near the computer center. Initially there was John, Steve Russell, Carol - a buxom Boston Irish secretary and I. Steve was a brillant programmer who had worked for John at Dartmouth (I believe). He was what we then called a 'programming bum' ...". The Students came one after another: P.W. Abrahams, D.G. Bobrow, R. Brayton, L. Hodes, L. Kleinrock, D.C. Luckham and J.R. Slagle (all were probably Ph.D. students in the departement of mathematics). D.M.R. Park joined the group in November.

We may get an idea of what McCarthy tought about his new programming language by reading the AI-Memo 1 which was written in the beginning of September 1958 at MIT. (We have a copy of a copy of N. Rochester's - who spent the year between September 1958 and September 1959 as visiting professor at MIT - which is hand-dated 16th September.) This memo (12) shows that McCarthy still had doubts about accepting the basic functions for list processing as proper functions. Other characteristics are the inclusion of conditional expressions, declarative statements and functional arguments without usage of the Lambda notation. Therefore we have problems with the real date - we have to accept a curious discrepancy between McCarthy's progress in connection with ALGOL and the lack of development in regard to LISP. Another source of doubt is the very fast progress in the next three months, extraordinary because ot the small amount of computer time the group had available.
The basic functions for list processing were still using the 3 bit parts of the IBM704 register, ther storage management was obviously planned after the model of IPL-V.
This paper is the most important document before the draft of the "Recursive Functions ..." (Memo8, March 1958). It is impossible to discuss it here in full extent; the interested reader is referred to (32). We restrict our remarks on the following. The memo had the title: "An Algebraic Language für the Manipulation of Symbolic Expressions". It presents the design of McCarthy's new language.

AN ALGEBRAIC LANGUAGE FOR THE MANIPULATION OF SYMBOLIC EXPRESSIONS 9/16/58 N. Rochester

by John McCarthy

Abstract: This memorandum is an outline of the specification of an incomplete algebraic language for manipulating symbolic expressions. The incompleteness lies in the fact that while I am confident that the language so far developed and described here is adequate and even more convenient than any previous language for describing symbolic manipulations, certain details of the process have to be explicitly mentioned in some cases and can be left to the program in others. This memorandum is only an outline and is sketchy on some important points.

I. Introduction

First we shall describe the uses to which the language can be put and the general features that distinguish it from other languages used for these purposes.

1.1. Applications of the language

1.1.1. Manipulating sentences in formal languages is necessary for programs that prove theorems and also for the advice taker project.

1.1.2. The formal processes of mathematics such as algebraic simplification, formal differentiation and

The language was planned to enable work with the following data structures: integers, machine word, truth values, locational quantities (i.e. labels) and functions (functional quantities). This reflected the unfortunate "quantity" concept of ALGOL. Lists were not regarded as data structures. A routine was envisaged for reading them into the machine. Therefore an external notation was defined. As elements of lists were permitted: numbers (integers and floating point numbers), symbols, text and again lists.
The proposed statements reflected FORTRAN in an obvious way. Besides the "arithmetical or replacement statement" the following kinds of statements were thought to be available: A GO-statements compound statements, statements for iteration, declarative statements (describing property lists) and statements for the definition of subprograms (and functions).
We should note that the assignment was enlarged in its applications by permitting function calls (terms) on its left side. The term was intended to select a part of a word in which the value of the right side was to be stored, for example: car(i) = cdr(j). The set of selector functions contained cwr, cpr, csr, cir, cdr, ctr and car. These functions for selecting parts of a word had as argument a 15-bit address. In contrast there was a similar group of functions which had full words as arguments (the functions pre, sgn, ind, dec, tag and add). Using the access function to words, cwr, there were relations between the two groups, for example: add(cwr(i)) = car(i). In addition to the possibility of using the assignment statement for modifying words a set of modification functions was planned: stwr, stpr, stsr,stir,stdr,sttr and star. (The last is called RPLACA today.)

The constructor functions were intended to create words: comb4(t,a,p,d) and comb5(t,a,s,i,d) had to put their arguments together and delivered the resulting word; consw, consel and consls fetched a word from the free-storage list, put their argument in it and result in the 15-bit address of the word filled in this way.

consw(w) was planned to have a full-word as argument, consel (a,d) to have two 15 bit addresses which were to be put in the address and decrement part of the word in the rest remaining empty), consls(a,d) worked in a similar way but filled the indicator field with a "2" - indicating a sublist. In addition there were other function which we do not describe here. We should mention only the functions for erasing list structures.
The first variants of copy and equal:

```
      function copy (J)
      /copy = (J=0 ⟶ 0, 1 ⟶ consw (comb 4(cpr (J), copy
   (cdr(J)), ctr (J), (cir (J) = 0 ⟶ car (J), cir(J) = 1
   ⟶ consw (cwr (car (J))), cir (J) = 2 ⟶ copy (car (J))))))
      \ return
      ¢ equal (L1,L2) = (L1 = L2 ⟶ 1, cir (L1) ≠ cir (L2)
   ⟶ 0, cir (L1) = 0 ∧ car (L1) ≠ car (L2) ⟶ 0, cir (L1)
   = 1 ∧ cwr (car(L1)) ≠ cwr (car(L2)) ⟶ 0, car(L1) = 2 ∧ ~
   equal (car(L1), car (L2)) ⟶ 0, 1 ⟶ equal (cdr(L1), cdr(L2))
```

As an further example of the notation of this early variant of LISP (the name was not yet invented) we present a function for differentiation which contains the maplist function with 3 arguments: The first argument is the list to be mapped, the second argument is the variable which has the various list elements as values and the third argument is the function body:

```
      function diff(J)
      diff = (ctr(J) = 1 ⟶ 0, car(J) = "x" ⟶ 1, car (J)
   = "plus" ⟶ consel("plus", maplist(cdr(J),K,diff(K))), car
   (J) = "times" ⟶ consel("plus", maplist (cdr(J),K, consel
   ("times", maplist(cdr(J),L,(L = K ⟶ diff(L), L = K ⟶
   copy (L))))))
      return
```

Without any doubt it is hard to imagine that this program could ever run correctly (this holds even if we correct the fifth line where the equal sign must be intended to be an unequal sign). However, McCarthy himself had this feeling, too. Nobody knows how many variants of this function he might have tried!
If we look at the memo2 (which is undated) then we get the impression that indeed the problems with this program have led to the inclusion of the Lambda-notation in LISP. This is somewhat surprising if we bear in mind the ideas of Mc Carthy in connection with ALGOL - where McCarthy proposed the Lambda-notation already in April or March. (We should mention that McCarthy himself was surprised to hear this. He had the opinion that only this example had have led him to the Lambda calculus. As we heve pointed out already this discrepancy might vanish if we had a better date for Memo1 (12).
It might be interesting to note McCarthy's arguments for introducing the Lambda-notation (in Memo 2) and changing the maplist function to have two arguments only (list and function): they are based on implementation considerations. McCarthy wrote that he tried to write a SAP-program (the assembly language of the IBM704) for maplist. He did not succeed in this: "The designation of J as the name of the indexing variable cannot conveniently be done in the calling sequence for maplist."
The imagination that the function which is delivered as argument is taken to be the machine instruction to call this function from inside of MAPLIST led him to the new definition and a satisfactory implementation. After using the Lambda notation the maplist definition was:

```
maplist(L,f)=(L=0→0,1→ consls(f(L),maplist(cdr(L)
             ,f)))
```

and the new version of the differentiation function:

```
      diff(L,V) = (car(L)= const⟶ copy(C0),car(L)= var⟶(car
   (cdr(L)) = V⟶ copy(C1),1⟶ copy(C0)),car(L)= plus ⟶
   consel(plus,maplist(cdr(L),λ(J,diff(car(J)♯,V)))),car(L)= times⟶
   consel(plus,maplist(cdr(L),λ(J,consel(times,maplist(cdr(L),
   λ(K,(J ≠ K⟶copy(car(K)),1⟶ diff(car(K),V)))))))))
```

Additionally the following detail should be noticed: in discussing the pecularities of this "algebraic language for the manipulation of symbolic expressions", McCarthy covered the algebraic notations and its advantages in particular, recursion and its implementation by means of push-down lists (which in fact were lists in the IPL-System of Newell, Shaw and Simon). He mentioned shortly that these expressions are repesentable by lists in an easy way. This remark is not restricted to data expressions ((12), p.2). Later in the paper (on page 11), McCarthy showed how a term $f(e_1,\ldots,e_n)$ is represented by a sequence of characters $(f,e_1,\ldots,e_n)$ and he remarked that it can be implemented as a list structure in such a way. Together with the programmatic statement concerning the intention to program the compiler for the language in the language itself (with a reference to the paper (11)), we might conclude that McCarthy had already developed his ideas of representing programs by list data structures.
The implementation began there after because the language seemed to be sufficiently specified. McCarthy and Minsky were eager to have a programming tool to realize their plans for artifical intelligence programs. The problem was that nobody in the group had a clear idea how one should write a compiler. The example of FORTRAN was rather discouraging because it required 30 person/years and this was extraordinary in those times.
To gain experience they startet to translate some of the functions into assembly language (SAP). To accomplish this they had to define conventions for calling functions, for working with push-down lists and for managing the storage. All this was done by the programmers Russell and Maling. We do not know when the students programmed their first function, did the translation or asked the programmers to translate it.
The input-/output-functions, the external standard notation for symbolic expressions, was fixed by such hand-translating. McCarthy used to speak of a "restricted spezialized external notation" - because of the limited set of characters on the typewriter and the keypunches - and he remarked that this notation was to be extended in future. He wrote (14): "... at present it seems that very little compromise will be required with the conventional notation beyond that required by the need to write expressions linearly with a limited set of characters."
Therefore the reason for selecting parantheses ("(and")") instead of brackets or braces was only a matter of chance because in 1958 the usual devices had only parantheses and nothing else. The prefix-

notation was induced by the list notation. The arguments were to be separated by commas.
In this Memo3(14), where McCarthy listed all conventions for the representation of numbers, symbols and lists; he also gived some hints concerning the internal representation of symbols and lists. The paper was closed with a section concerning a revision in the system described in the earlier memos. "Some experience in programming in the system and hand-compiling the resulting programs suggest some changes in the set of elementary functions.

1. The functions which refer to parts of the word other that the address and the decrement can be omitted.
2. The functions referring to whole words are retained but will be used only inside property lists.
3. The distinction between consel and consls is abolished so we will call the new function cons.
4. The storage and pointer functions have not been used so far and hence are tentatively dropped."

That means the ancestors of RPLACA and RPLACD (star and stdr) were killed at 10/21/58. McCarthy described then in an appendix the functions add, dec, comb, car, cdr, consw, cons, erase, copy, equal, eralis, maplist, print and read. We give here copy and equal for contrast with the same functions given in the Memo1:

```
copy(L) = (L=0→0,car(L) = 0→L,1→cons(copy(car(L)),copy(cdr(L))))
equal(L1,L2) = (L1)L2→1,car(L1) =0∨car(L2) = 0→0,
      1→equal(car(L1),car(L2)) equal(cdr(L1),cdr(L2)))
```

It turned out that the realization of recursive function was the most complicate problem in translating the LISP-programs into SAP-routines. The importance of this problem for LISP is outstanding even today and it was taken seriously in 1958 (without very much programming experience). For solving this problem it was possible to use the example of IPL. However, it seemed to McCarthy that implementing stacks as lists was not appropriate and he decided to use linear sequences of machine registers for a speed-up.He first thought of associating a private stack with every functions, but soon changed his mind. A central stack, the "public push-down list", remained.

The function-calling conventions were defined before end of October 1958. McCarthy reported about it in Memo4. This memo is important because it is the first written document where the name "LISP" was used (presumably end of Ocotber 1958). The function call was executed in such a way that the necessary registers (words in storage) were saved by a SAVE-routine to be accessible by the function during its work. Each of these blocks in the stack was later extended to contain the name of the function, size of the block and its origin in the storage. Corresponding to these conventions for function entry the exit from functions was specified (UNSAVE-routine).
To a large part this memo contains examples for hand translated functions, i.e. SAP-code for cons, copy, maplist, diff anf further new functions (select, list, eql, cpl, search, subst, error, pair).

The end is again very important for the development of LISP: a new kind of function, the "substitutional functions" were introduced. We cite the original text: "The value of a substitutional function applied to a list of arguments is the result of substitutions these arguments for the objects on an ordered list arguments in a certain expression containing these arguments ... There is a routine apply (L,f) whose value is the result of applying a function to a list of arguments. This function expects the function f itself to be described by an expression. The kinds of expression for functions which apply will interpret has not been determined and for the present we shall only consider the case where car (f)=subfun. Thus our initial version of apply is:

```
apply(L,f)=(car(f)=subfun→sublis(pair(car(cdr(f)),L),
          car(cdr(cdr(f)))),1→error)"
```

This means that McCarthy did use the word "argument" in the above sentence for variables and arguments simultaneously (formal parameter,actual parameter). If we exchange the first list element then we have here genuine Lambda-expressions:

```
(subfun(x)(cons x 0))=(LAMBDA(X)(CONS X NIL))
```

There is an example (15,p. 21) where this function **apply** is used in a more complicated diff-functions to fetch partial derivatives (gradients) from the P-lists of the corresponding function symbols.

It is hard to believe that it was this apply function which caused the famous dialogue between McCarthy and Russell. Later we describe this event. On the other side - if we do not want to accept this function as a general evaluation function (and there is much lacking) then it is of course an original kernel. If this function is running then the step of including the application of implemented functions to arguments is definitely not a big step - and we should here bear in mind the maplist-function. If we take this point of view, we lose the ability to imagine the famous communication.

It is a pity that this memo has no date. Therefore we have to take the Memo5 (30), written by N. Rochester, into account. It is dated 11/18/58 and sets an upper bound. It should be clear that this apply function can be extended easily into a complete interpreter. If we have defined functions like maplist, pair or sublis (and implemented them) then the apply function should not make further problems. Therefore we understand now D. Park well when he reports that in the middle of November when he joined the AI project there was a running LISP interpreter ...

Memo5 is a remarkable paper. Rochester gived in it his thoughts for developing LISP further tried to attack the differentiation problem of McCarthy and then presented what is the first symbolic simplification program ever written. Rochester gived LISP-definitions of various functions (including Mc Carthy's subst and additionally subfp, insert, canceladdend, cancelfactor, multal ect.) and called for a basic function = (which was used implicitly by McCarthy) and proposed rules for the two-dimensional notation for LISP. These rules concern conditional expressions - where McCarthy gives unstructured expressions which are hard to understand - and introduce identation. Other rules concern composed functional expressions and propose to write instead of

```
F(G(Z))   either   G(Z)          or F(below)
                   F(above)         G(Z)
```

Rochester proposed the revival of the storage functions in the shape of a REPLACE-function which definitely is the father of RPLACA and RPLACD. Of lasting consequence is his proposal to write a short notation for compound car- and cdr-calls. We should

mention here that McCarthy in his Memo4 always uses single functions and so does Russell in Memo6. In Memo8 McCarthy abandoned car and cdr completely and used first and rest instead. In the second version of the "Recursive functions ..." - paper (April 1959) we find functions like caar etc. .
Of course, this does not force us to assume Rochester invented this notation.
In the rest of Memo5 Rochester gived the functions for the simplify program. The main function is simp, its subfunctions are simplus, simptimes, simpminus which are applied consecutively to the expression.
We restrict ourselfes to simp and simplus:

```
simp(J)=simpplus(simpminus(simptimes(J)))
simpplus(J)=
(J=0→0,
 car(J)=0→J,
 car(J)=plus→(
      cdr(J)=0→CO,
      cdr(cdr(J))=0→simplus(cdr(J)),
      1→simpaddend(J,cdr(J))),
1→cons(simplus(car(J)),simplus(cdr(J))))
```

At the end of Novmeber 1958 McCarthy was in England where he gave his "Advice-Taker" talk. We already have used this paper for our purposes.
Investigating the early history of LISP it is a major obstacle that no important memo was written between the end of November 1958 and the beginning of March 1959. Memo6 was written by S. Russell and is related to the programming and debugging of SAP-routines for including them into the system; Memo7 is due to McCarthy and discusses elements of LISP-compilation. In just this time the crucial events must have happened.
We repeat here the best version which is known today (even if we - as it was stressed - have doubts): In proving that using LISP one is able not only to describe the computable functions in a simpler way than for example with the formalism of Turing-machines but also that the same class of functions is defined, McCarthy chose the problem of programming a general (universal) LISP-function and to showing that it is shorter and more understandable than the description using a universal Turing-machine. (A student had to write a Turingmachine program in the autumn. As should be known a universal Turingmachine is characterized by its ability to accept a description of another Turingmachine and data for it on ints inputtape and to imitiate the operation of this other Turingmachine.) In consequence an universal LISP-function hat to accept the description of another LISP-function and in addition some arguments for it as a parameter and to deliver the same result as the argument function would have delivered.
To accomplish this McCarthy had to find a notation in which every LISP-expression and LISP-function could be expressed as a symbolic expression. It occured to him to use the list notation. This is obvious in the case of functional terms. Only symbols and list-constants made trouble, because these could be used in programs and must not be exchanged by mistake with function names or translated terms. But inventing the quotation McCarthy could overcome this problem. We do not know which translation rules McCarthy used it the end of 1958 - we can only look at later versions.
As far we know the first universal function was indeed apply - eval was not present at all or not seen to be important. When McCarthy was working on this function S. Russel saw it and suggested translating it by hand - as he had done so often - and adding it to the program system. McCarthy recalls (22): "... This EVAL was written and published in the paper and Steve Russell said, look, why don't I program this EVAL and you remember the interpreter, and I said to him, ho, ho, you're confusing theory with practice, this EVAL is intended for reading not for computing. But he went ahead and did it. That is, he compiled the EVAL in my paper into 704 machine code fixing bugs and then advertised this as a LISP interpreter which it certainly was, so at that point LISP had essentially the form that it has today, the S-expression form ..."

If we take the semi-annual report of the computation center of MIT of December 1958 (17) then we read that the work to debug the interpreter was underway at this time: "At present, routines written in LISP are hand-translated into SAP, but we expect to begin on a compiler soon.A routine for applying a function to an argument, where the function is described by a symbolic expression, has been programmed but not yet debugged. This routine will be the basis of an interpreter."

For the sake of completness we should mentioned that McCarthy himself was writing a Memo5 in which he planned to described the functions conc, subfp (with reference to Rochester) and adj. This memo was never completed.

With regard to the interval between December 1958 and March 1959 we can only speculate at present. Presumably an interpreter was tested - but what interpreter is unknown.
The next memo, Memo8, is dated 4th of March 1959. It is already called "Recursive Functions of Symbolic Expressions and their Computation by Machine". It is structured in a very similar way as the later version published in the Communications of the ACM. It contains sections on S-expressions, elementary functions and predicates (FIRST instead of CAR, REST instead of CDR), functional expressions and functions which are compositions of elementary functions, conditional expressions, definitions of recursive functions, functions with functional arguments, labelled expressions, computable functions. In the section "LISP self-applied" we might find the first known version of the translation rules from LISP into S-expressions:

rule 1: quotation of constant S-expressions.
rule 2: translation of variables and function-names into symbols.
rule 3: translation of terms into lists.
rule 4: the empty S-expression is called NIL.
rule 5: the truth-values 0 and 1 are translated into F and T, respectively. Conditional expressions $[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots; p_k \rightarrow e_k]$ are translated into $(COND,(p_1,e_1),(p_2,e_2),\ldots,(p_k,e_k))$
rule 6: $\lambda[[x;\ldots;s];\varepsilon]$ is translated into (LAMBDA,(X,...,Z);ε).
rule 7: label $[\mathcal{a};\varepsilon]$ is translated into (LABEL,$\mathcal{a}$,ε).
rule 8: x=y is translated into (EQ,X,Y).

We must add at his point that the label-expression was introduced by Rochester for writing recursive Lambda-expressions. This is mentioned by McCarthy in (21). If we look at Memo5 then Rochesters pro-

posal was to write: λ(F(K),body-with-K). It is well known now that other constructs of combinatorial logic could have been used to avoid it.

We want now to concentrate on the interpreter. Even though S. Russell disapproved of this, we see good reason for accepting this program as the apply-eval-functions designed in the December of 1958. The first reason is that if McCarthy would have written down an earlier version then some trace should have survived. The second is that the early APPLY (which was given in Memo4) was also based on substitutions. A transition to bindings seems to be of such a significance that McCarthy would not have undone it in a later paper.
McCarthy gives the following definitions:

```
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]]]
evcon[c] = [eval[first[first[c]]] = 1 →eval[first[rest[first[e]]]];
            1 →evcon[rest[c]]]
evlam[vars;exp;args] = [null[vars] → eval[exp];
                        1→ evlam[rest[vars];
                                 subst[first[args];first[vars];exp];
                                 rest[args]]]
```

After a little thought we form the impression that this program will not run correctly. The main reason is the assumed wrong substitution which substitutes arguments for variables at any place: in correct positions in terms but in incorrect positions in variable-lists and quoted S-expressions.Therefore we are not very happy with the procedure for substituting the variables consecutively in the body of the Lambda-expression: In sub-expressions which occur after substituting unevaluated arguments for earlier variables the values (unevaluated arguments again) of later variables will be substituted and this is hardly intended. The reason for this error might be that the previous version was for functions with one argument only.
If we imagine a correct substitution function and accept the necessity for quoting all constants then there remains the well known problem of free variables moved to wrong environments because of the lazy evaluation. If McCarthy had added a case for first(e)=LAMBDA and constructed another Lambda-expression with renamed variables this would have been fixed.

There is an addendum dated the 13th of March which shows that McCarthy detected the substitution problems (and the wrong "1" in evcon). He proposed the following new substitution function:

```
subsq=λ[[x;y,z]; [null [z]→Λ
                  atom [z]→ [ y=z →x;1 →z];
                  first [z]=QUOTE →z;
                  1 →combine [subsq [x;y;first [z]];
                   subsq [x;y;rest [z]]]]]
```

This solves the problem in conncection with quoted subexpressions but not the problem related to variable list. We suspect that McCarthy simply had forgotten the functional arguments - a mistake of lasting importance. Therefore either the translation rule for Lambda-expressions is wrong ot there is a missing case in the eval-function. We all know that somebody - perhaps S. Russell - later decided that the translation rule for Lambda-expression was to be changed to deliver quoted Lambda-lists in this case. This was not a well thought out decision and later induced many further problems.
If we analyze the text further we cannot understand why the eval-function is used at all. The sublanguage of LISP it evaluates contains only terms (or as Mc Carthy used to call them "forms") and with the exception of F and T no atoms of any kind. There is no problem if we replace both by (QUOTE F) and (QUOTE T) respecively. Therefore we might replace any recursive call of eval by a call of apply.

We believe that the fact that this function contains errors which should have been found after some test runs proves again that this is the first LISP interpreter. If we take into account that McCarthy produced a correction only after a week we come to question the assumption that this was hand translated already in December. Errors of this magnitude should have been removed weeks ago. An argument against the December assumption as possible date of the first interpreter is the experience that a paper of this size and complexity can not be written in a few days.
As far as we know S. Russell was handtranslating the interpreter while McCarthy was cleaning up his paper. We do not know when and why McCarthy invented bindings as a surrogate for the substitution. It is interesting that he did not start to change the evaluation style into a call-by-value scheme. Of course, a probably reason why substitution was removed is the huge amount of copying work which is necessary to accomplish it.
The next paper which tells us something about the development of the LISP-interpreter is the Quarterly Progras Report of the Research Laboratory for Electronics in April 1959 (19). In the general outline the paper is reviewed thoroughly. The functions car, cdr and their compositas are used now. The translation rules are corrected to reflect their recursive application. Dot notation and dotted pairs are introduced.
Before we look to the general remarks reflecting the state of the system we want to present the new version of the interpreter:

```
    The S-function apply is defined by
    apply[f;args]=eval[cons[f;appq[args]];NIL]
where
    appq[m]=[null[m]→NIL;T→cons[list[QUOTE;car[m]];appq[cdr[m]]]]
and

        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]]
```

```
and
    evcon[c;a]=[eval[caar[c];a]→eval[cadar[c];a];T→evcon[cdr[c];a]]
and
    evlis[m;a]=[null[m]→NIL;T→cons[list[QUOTE;eval[car[m];a]];
evlis[cdr[m];a]]
```

As we see the interpreter still works with unevaluated arguments. The eval-function now "has two arguments, an expression e to be evaluated, and a list of pairs a. The first item of each pair is an atomic symbol, and the second is the expression is the expression for which the symbol stands." (19, p. 135) Eval accepts symbols and lists - no numbers. The truth values had to be quoted - as we proposed above. It is interesting to note that the eval by evaluating a symbol searches in a list of pairs (the terminus "association list" was reserved at this time for the symbols component we call p-list today) and evaluates the identified value using the same p-list again. Therefore naming conflicts of variables would have a catastrophic effect. The handling of functional arguments is as erroneous as before.

As McCarthy tells us he wrote this version of the RFSE a very short time before its publication. Therefore the report describes the state at the begin of April very exactly: "At the present time ... approximately 20 subroutines have been hand-compiled and checked-out. These include routines for reading and printing S-expressions ... In particular, apply and its satellites are working. An operator program for apply permits the user to evaluate any S-function with any set of S-expressions as arguments without writing any machine language program: apply then serves as an interpreter for the programming system. Work has started on a compiler that will produce SAP language routines from the S-expressions for S-functions. Two drafts of the of the compiler have been written in LISP." (19, p.144)

On another page McCarthy mentions the plan to enable sequential (instruction-oriented) programming: "In addition to the facilities for describing S-functions, there will be facilities for using S-functions in programs written as sequences of statement along the lines of FORTRAN or IAL ..." (27, p. 139)

A further important concept which is mentioned in the paper is the garbage collection idea under the heading "Free-Storage List". In this section McCarthy describes the way how free registers are managed. The difference to the IPL-solution is condensed in the sentence: "No provision need be made for the user to program the return of registers to the free-storage list. The return takes place automatically ...". If we look at the short and elegant functions for copying or applying a function to every element of a list then we become aware that an explicit erasing of list structures would have obscured the notation. McCarthy had provided functions for list erasure but never really used them. This worked for a while but when the interpreter was implemented the situation must have happened that all registers were exhausted. To overcome this bottleneck, the idea of garbage collection emerged. As fas as Abrahams and Maling remember, R. Sivler first came out with this proposal. However, McCarthy - after a discussion with R. Silver, convinced us that he conceived this.

In (19), McCarthy describes the idea as follows: "Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts ...". Then he gives a description of the now well known steps of garbage collection. He concludes: "This process, because it is entireley automatic, is more convenient for the programmer than a system in which he has to keep track of lists and has to erase unwanted lists. Its efficiency depends upon not coming close to exhausting the available memory with accessible lists. This is because the reclaming process requires several seconds to execute, and therefore must result in the addition of at least several thousand registers to the free-storage list if the program is not to spend most of its time in reclamation."

This was pure theory then because the first garbage collector was implemented by Dan Edwards during June/July of 1959. At the end of April Dan Edwards was one of the first users of the LISP-system.

We have to make it clear that the developing LISP-implementation already had serious users at this time. In the middle of March 1959 D. Arden, N. Rochester, J. Slagle, a P. Markstein and S.H. Goldberg met and discussed the application of LISP to problems of formula manipulation (we call it now "symbolic algebraic computation"). They had selected the problem of symbolic analysis of electrical networks. Goldberg was appointed to cover the whole subject in his masters thesis, other students had to do partial work. Two of these students were then D.J. Edwards, who took over the reduction of the symbolic matrices, and S.Z. Rubenstein, who had to program the construction of the symbolic admittance matrix.

The Bachelor Theses of the latter two, which they delivered at the end of May , tell us much concerning the state of the LISP-system in May 1959. One amazing point is the used method to write sequential programs: in those times (and aren't there many people who think this way even today?) nobody could envisage programming without planning sequences of actions. However, the LISP-system at this time contained none of the functions like PROG or PROGN - and of course there was no implicit PROGN neither. Therefore the programmers misused conditional expressions because of the sequential execution of the predicates. As long as the predicates yielded NIL a sequence of actions could be simulated. Therefore some of the pseudo-functions (evaluated for side-effect mainly) were defined to deliver the value NIL to allow this style of programming (this was valid for RPLACA and RPLACD, for example).

In Rubinstein's thesis is a reference which could not be verified until now. Rubensein referred to a "programmers manual", where, in its 10th version, S. Russell had described the universal function APPLY. We do not know of such a manual. The manuscript - hand-written by McCarthy - which might be declared to be the first draft of a manual was certainly written in October 1959.

By chance one early version of the LISP-interpreter - dated 5/15/59 - is saved at MIT (in the possession of J. Moses) which does not contain the garbage collector. We cannot give here the SAP listing of course but the comments are written in LISP and it is interesting to compare them with our last version:

```
APPLY(F,L,A) = SELECT(CAR(L),-1,APP2(F,L,A),
                         LAMBDA,EVAL(F,APPEND(PAIR(CADR(F),L),A)),
                         LABEL ,APPLY(CADDR(F),L,APPEND(PAIR1(CADR(F),(CADDR(F)),A))
                         APPLY(EVAL(F,A),L,A)).
APP2(F,L,A) = SELECT(F,CAR,CAAR(L),
                     CDR,CDAR(L),
                     CONS,CONS(CAR(L),CADR(L)),
                     LIST,COPY(L),
                     SEARCH(F,LAMBDA(J,CAR(J) = SUBR OR
                                     CAR(J) = EXPR)
                            LAMBDA(J,CAR(J) = SUBR YIELDS APP3(CWADR(J),DISTRIB(L)),
                            1 YIELDS APPLY(CADR(J),L,A)))
                           ERROR))
EVAL(E,A) = SELECT(CAR(E),-1,SEARCH(A,LAMBDA(J,CAAR(J) = E),
                                    LAMBDA(J,CADAR(J)),ERROR),
                       COND,EVCOND(CDR(E),A),
                       SUB,SUBLIS(A,EVAL(CADR(E),A)),
                       CONST,CADR(E),
                       LABEL,EVAL(CADDR(E),APPEND(PAIR1(CADR(E),CADDR(E)),A)),
                       VARC,SEARCH(A,LAMBDA(J,CAAR(J) = CADR(E)),
                                   LAMBDA(J,CADAR(J)),ERROR),
                       VARE,SEARCH(A,LAMBDA(J,CAAR(J) = CADR(E)),
                                   LAMBDA(J,EVAL(CADAR(J),CDR(J))),ERROR),
                       APPLY(CAR(E),MAPLIST(CDR(E),LAMBDA(J,EVAL(CAR(J),A)))A))
```

As we can see this interpreter worked with evaluated arguments (last clause of eval). However, there are two kinds of variables: One with evaluated associates on the a-list and another with unevaluated. A specialform (compare the later SPECIAL!) with function name varc resp. vare is responsible for this. We note again a missing case for LAMBDA in eval. A case for LABEL is included!

Because of place considerations we must end our course in history here. For later events, the interested reader is referred to (32).

This research could not have been done without the help and encouragement of J. McCarthy. Many other people have contributed comments and hints. They are listed in (32). We want to acknowledge the remarks of R. Bell and H. Wedekind which made this paper more readable.

Literature:

(1) D.J. Edwards: Symbolic Circuit Analysis with 704 Electronic Computer, MIT BS Thesis, Dept. of EE. Cambridge 1959.

(2) H. Gelernter, J.R. Hansen, C.L. Gerberich : A FORTRAN Compiled List Processing Language. Journal ACM 7(1960)2,87-101.

(3) H. Gelernter, J.R. Hanse D. Loveland: Empirical Explorations of the Geometry Machine. Proc. WJCC 1960.

(4) H. Gelernter, N. Rochester: Realization of a Geometry Theorem Proving Machine. Information Processing, Proc. ICOI Paris 1959, München 1960.

(5) S.H. Goldberg: Solution of an Electrical Network Using a Digital Computer. MIT MS Thesis, Dept. of EE. Cambridge 1959.

(6) J.R. Hansen: The Use of the FORTRAN Compiled List Processing Language. IBM Th. Watson Res. Ctr., Resp. Rep. RC-282, Yorktown Heights, Juni 1960

(7) IBM: Preliminary Report: Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN. IBM Corp., Progr. Res. Group, Appl. Sci. Div., 1954.

(8) IBM: The FORTRAN automatic Coding System for the IBM704 EDPM, Programmers Manual. IBM Corp., C28-6000, 1956.

(9) J. McCarthy: Notes on the Sky-blue Compiler. On Functions which Produce more than one Quantity as Output. Letter to the director of MIT Computer Center, Cambridge 7/11/57.

(10) J. McCarthy: The Programming Problem. Handwritten Note. Probably 1956.

(11) J. McCarthy: Some Proposals for the Volume 2 (V2) Language. Letter to A.J. Perlis and W. Turanski, 6/1958.

(12) J. McCarthy: An Algebraic Language for the Manipulation of Symbolic Expressions. MIT AI Lab., AI Memo No. 1, Cambridge Sept. 1958.

(13) J. McCarthy: A Revised Version of "MAPLIST". MIT AI Lab., AI Memo No. 2, Cambridge Sept. 1958.

(14) J. McCarthy: Symbol Manipulating Language - Revisions of the Language. MIT AI Lab., AI Memo No. 3, Cambridge October 1958.

(15) J. McCarthy: Symbol Manipulating Language - Revisions of the Lanuage. MIT AI Lab., AI Memo No. 4, Cambridge October 1958.

(16) J. McCarthy: The Advice Taker, a Program with Common Sense. Symp. on the Mechanization of Thought Processes. Nat. Phys. Lab., Teddington (England) 24.-27.11.1958.

(17) J. McCarthy: Artificial Intelligence Project. In: Progress Report No. 4 of the Research and Educational Activities in Machine Computation by the Cooperating Colleges of New England, Dec. 1958 (Semi-Annual Report).

(18) J. McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. MIT AI Lab., AI Memo No. 8, Cambridge March 1959.

(19) J. McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. In: (27)

(20) J. McCarthy: LISP Programmers Manual, Handwritten Draft, MIT AI Lab., Cambridge Oct. 1959.

(21) J. McCarthy: Recursive Functions of Symbolic Expressions and their Computation by Machine. Communications ACM, 3(1960)3, 184-195.

(22) J. McCarthy: LISP History. Talk at MIT, Spring or Summer 1974 (Written from tape 7/10/75, unpublished)

(23) J. McCarthy: History of LISP. ACM SIGPLAN Notices, 13(1978)8, 217-223.

(24) J. McCarthy: Personal Communication 1976.

(25) J. McCarthy et al.: A Proposal for the Dartmouth Summer Project on Artificial Intelligence. 8/31/55.

(26) J. McCarthy, M. Merwin: Routines for Turning FORTRAN Program into FORTRAN Routine Computation Center MIT, Cambridge, January 1958.

(27) J. McCarthy, M.L. Minsky: Artificial Intelligence. Quarterly Progress Report No. 53, Res. Lab. of Electronics MIT, Cambridge, April 1959.

(28) J.W. Backus et al.: Proposal for a Programming Language. ACM April/May 1958.

(29) P. McCorduck: Machines who Think. W.H. Freeman & Comp., San Francisco 1979.

(30) N. Rochester: Symbol Manipulation Language. MIT AI Lab., AI Memo No. 5, Cambridge November 1959.

(31) S.Z. Rubenstein: The Construction of the Admittance Matrix with a Digital Computer. MIT BS Thesis, Dept. of EE. Cambridge, May 1959.

(32) H. Stoyan: LISP - Anwendungsgebiete, Grundbegriffe, Geschichte. Akademie Verlag, Berlin 1980.

(33) H. Stoyan: LISP Compilation Viewed as Provable Semantics Preserving Program Transformation. LNCS 162: J.A. van Hulzen Computer Algebra, Springer Verlag, Berlin etc. 1983.

(34) H. Stoyan: Maschinen-Unabhängige Code-Erzeugung als semantikerhaltende beweisbare Programmtransformation. Habil. Thesis, University of Erlangen-Nürnberg, Erlangen 1984.

(35) J. Sammet: Programming Languages: History and Fundamentals. Englewood Cliffs 1969.