The History of LISP

Herbert Stoyan (translated by ChatGPT and Paul McJones)

1980; updated in 1990s?; translated December 2023

Contents

1 Introduction **3**

2 The Emergence of LISP **3**

3 Local distribution of LISP 1962–1966 **31**

4 Worldwide distribution of LISP 1966–1978 **41**
4.1 MacLISP . . . 43
4.2 InterLISP . . . 48
4.3 STANFORD LISP 1.6 . . . 53
4.4 Other LISP implementations in the USA . . . 54

5 LISP in Europe **63**
5.1 United Kingdom . . . 63
5.2 France . . . 66
5.3 Netherlands . . . 68
5.4 Norway . . . 71
5.5 Sweden . . . 73
5.6 Federal Republic of Germany . . . 75
5.7 German Democratic Republic (DDR) . . . 78
5.8 Poland . . . 80
5.9 USSR . . . 81
5.10 Hungary . . . 83
5.11 Italy . . . 84
5.12 Denmark . . . 85
5.13 Czechoslovakia (CSSR) . . . 85
5.14 Belgium . . . 85
5.15 Switzerland . . . 85

6 LISP Overseas **86**
6.1 Mexico . . . 86
6.2 Japan . . . 87
6.3 Canada and Australia . . . 89
6.4 India . . . 89

7 Metasystems **89**
7.1 Classification . . . 89
7.2 Before LISP 2: Pattern Matching and System-Independent Input 90
7.3 LISP 2 . . . 92
7.4 Between LISP 2 and PLANNER: CONVERT, FLIP, MLISP, and Formula Manipulation Systems . . . 96
7.5 PLANNER: Backtracking, Pattern Matching, and Deductive Mechanisms . . . 100
7.6 QA4 . . . 104
7.7 MLISP 2 . . . 107

7.8 From PLANNER to CONNIVER 110
7.9 QLISP, CLISP, CGOL, and RLISP 113

1 Introduction

LISP is intended as a programming language for communication between humans and computer systems. However, programs need to be conceived and developed, programmers discuss them, and use them to exemplify certain ideas. In many cases, programs are never translated into machine-readable form; they are important as programming attempts, as approaches to understanding a new problem.

For these reasons, it can be inferred that every programming language is a social phenomenon. Characterizing it solely as a set of permissible character strings and addressing only questions of its recognition, processing, and description is by no means sufficient.

It has always been a good practice to analyze the history of a social phenomenon when studying it. This reveals the necessary and incidental reasons for its existence, allowing consideration of its impact on human society. In the case of an object like a programming language, created for human use, one can also attempt to determine the extent to which the finished product has proven itself and how, over time, changes or redesigns have occurred due to new demands and accumulated knowledge about the properties of this object.

Certainly, one may be able to neglect the pragmatic and, more broadly, the social component in many important and entirely justified scientific works on programming languages. However, anyone who thinks they can develop a programming language without these considerations should not be surprised if their language is scarcely used.

The history of a programming language, which must encompass not only its original development but also its dissemination, use, and evolution, is, of course, an essential part of the history of computer science. The attempt undertaken here to write the history of the LISP language is thus a contribution to the recent history of science.

2 The Emergence of LISP

The programming language LISP is the response of a man, John McCarthy, to the formulation and notation problems for programs at the end of the 1950s. McCarthy has certainly been in close discussion with a circle of friends and acquaintances (among them, Marvin Minsky is already worth mentioning), and so some detailed solutions may come from others – in some cases, this is explicitly known. However, to be able to consider the background of LISP, one must look at McCarthy's activities during that time.

McCarthy, born in 1927, studied mathematics and obtained the academic degree of Bachelor of Science from the California Institute of Technology (Caltech) in 1948. In 1951, he earned his doctorate with a thesis on differential equations at Princeton University.

The interest in questions about how to create intelligent cybernetic systems came to him when he attended a symposium on cybernetics at Caltech in 1949

and heard, among others, John von Neumann. Thus, he dealt with similar problems on the side and wrote a paper on the inversion of Turing machines, in which he presented some of his views [MCC56a]. This article, completely unnoticed in its significance according to Minsky, appeared in 1956 in Automata Studies, a collection edited by McCarthy and C.E. Shannon at Princeton University.

A more intensive engagement began when Shannon invited McCarthy to the Bell Telephone Laboratories in the summer of 1952. At that time, Minsky, who was still studying in Princeton, was also there. Together, they worked on questions concerning the relationship between cybernetics and algorithm theory.

When McCarthy became Assistant Professor of Mathematics at Dartmouth College in Hanover, New Hampshire, in 1956, he was still on friendly terms with both of them. By then, they had come to the realization that electronic computers, rather than cybernetic models and machines, were better suited for achieving machine intelligence. They had also heard from other researchers in this field who had come to similar conclusions in different cities across the USA. Thus, they had the idea to invite everyone to a working summer at Dartmouth College, called the Dartmouth Summer School on Artificial Intelligence. In a request for support to the Rockefeller Foundation, McCarthy wrote about "a two-month study by 10 men... The study is to be based on the supposition that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" [MCCO77].

Although no one expected the group to fulfill their ambitious goals (according to McCarthy, the support request would still make sense today, with some names, dates, and amounts of money changed), they got the funding, and the summer could begin.

Among the participants—besides the three mentioned organizers, N. Rochester from IBM, O. Selfridge and R. Solomonoff from MIT Cambridge, and T. Moore and A. Samuel again from IBM—there were also two who already had experience in using electronic computers: A. Newell and H. Simon. Other interested individuals joined for a short time, and it is amusing to hear that one of them was A. Bernstein, who was working on a chess program that soon became publicly known, much to the displeasure of his company, IBM. IBM feared that potential buyers would be hesitant to acquire overly intelligent computers [MCCO77].

Newell and Simon had developed programs for proving and manipulating logical theorems on the JOHNNIAC computer at RAND Corporation. Since the importance of hardware was consistently overestimated at the time, they referred to their system as the "logic theory machine"—today, they would have spoken from the outset of a new programming language. However, they had already recognized that a formulation medium was necessary in such a system. The logical language LL in their information-processing system was to be equipped with capabilities for list processing—"one of the most important ideas ever to emerge in programming" (J. Sammet [SAM69]).

IPL (Information Processing Language), as it was then called, had already been developed to a considerable extent by the summer of 1956—presumably, work on the implementation of IPL II was underway—allowing the authors to present it at conferences and discussions [NEW56].

Due to its low (i.e., machine-level) nature ("...derived from the format of a loader for the JOHNNIAC, which was available to them..." [MCC78a]), IPL did not elicit unanimous enthusiasm from those present. In the spring of 1956, reports about IBM's formula translator, with the FORTRAN language having progressed beyond its development stage [FOR54] and being applicable [FOR56], had been heard. Although McCarthy, a visitor at IBM, had only loose contact with this development, he was impressed by the fundamental idea of writing programs using algebraic means, i.e., mathematical expressions. All that was known was that artificial intelligence would require working with symbolic expressions—numerical calculations would have little significance. It was easy to imagine that any subexpression could be obtained very conveniently by composing functions that deliver the direct parts.

Nevertheless, IPL had a significant impact. In addition to list structures, the programmability of recursive programs should be mentioned, although the implementation technique was not yet well-developed at that time. Stack storage was represented by lists, leading to very slow call mechanisms. However, they were used for the first time here.

Other contributions of IPL to programming technology are summarized by Sammet [SAM69][p. 400]: "The hierarchical subroutine implementation is simpler than in FORTRAN or ALGOL... Attribute-value lists proved to be an important data structure... IPL introduced a concept and secure means for dynamically extending data areas as needed during program execution...". The tracing facilities offered by IPL may also have been significant for later LISP.

However, in 1956, IPL was little more than a loose collection "of commands at the assembly language level for list processing" ([SAM69], p. 383). IPL II [NEW57a, b] was implemented on a JOHNNIAC at the RAND Corporation. IPL III and IPL IV represented intermediate steps toward IPL V, which was available at the end of 1957. An implementation on an IBM 650 at the Carnegie Institute of Technology took place in early 1958, and the final version was on the IBM 704 by the summer of 1959 [NEW60]. IPL V was one of the most implemented languages at that time ([SAM69], p. 389).

Although McCarthy was not drawn to the external form of the language, he was very impressed by the realization. The list structures—often referred to as NSS-memory at the time by Newell, Shaw, and Simon, the authors of IPL, see, e.g., [GE60a.]—seemed very flexible and highly suitable for accommodating complex logical-symbolic information. Given this evaluation of IPL at the time, P.W. Abrahams' recollection "The original design of LISP actually comes from Newell's and Simon's IPL. In the first few memos that McCarthy wrote on this subject, it was very similar to IPL but with some new ideas" is not correct or at least misleading.

One thing is certain: McCarthy had been thinking about a formulation medium since mid-1956, and FORTRAN had very attractive features for him. Among these, it seemed particularly important to him that in FORTRAN [FOR56], function expressions could be nested, allowing for expressing a quite complicated transformation step in a single line. It is entirely possible that by 1956, he was already thinking of introducing new functions in this way. Mc-

Carthy, possibly with information from IBM, was ahead of his time here, as only FORTRAN II [FOR58] had the Function Statement, which had to be noted before the actual program and allowed for such function definitions. This was announced in early 1957 [B57].

McCarthy's considerations regarding any potential realization were quite early on tied to the IBM 704. There were two reasons for this: IBM wanted to establish a computing center for New England at MIT Cambridge, and it seemed like IBM wanted to generously support research in the field of Artificial Intelligence. So, probably in early 1957, prompted by Rochester, who was then the head of the Information Research Department at the Poughkeepsie Research Center, an initiative was taken to realize an idea expressed by Minsky at the summer conference: to develop a program for proving geometric statements.

H. Gelernter, who had just completed his Ph.D. and had been working at IBM since the fall of 1956, became interested in this project, and McCarthy was to serve as a consultant. The connection seems to have been quite close at first, as long as the fundamental decisions about language and implementation were under debate. Later, when the actual program for proving geometric theorems was tackled, the contact was extremely loose.

To represent symbolic information in the machine, such as the geometric statement that two triangles are congruent, list structures were to be used. Thus, McCarthy could contribute his ideas developed at Dartmouth: he suggested using FORTRAN for the programming work. "...there were certain considerations regarding the conversion of JOHNNIAC IPL to the IBM 704 computer. However, McCarthy, who advised the project, recommended that FORTRAN could be adapted to serve the same purpose. He explained that the nesting of functions allowed in FORTRAN made it possible to construct complex information-processing subroutines with a single statement. Since then, the authors have discovered another significant advantage of an algebraic list-processing language like the one built within the framework of FORTRAN. It is the close analogy that exists between the structure of an NSS list[1] and a certain class of algebraic expressions that can be written in the language..."[GE60aq][p. 88].

Most proposals regarding how to represent list structures in the memory of the IBM 704 and which basic functions were suitable came from McCarthy. A word (register) of the IBM 704 consisted of five parts: sign, prefix, decrement, tag, and address.

[1] see footnote on p.136

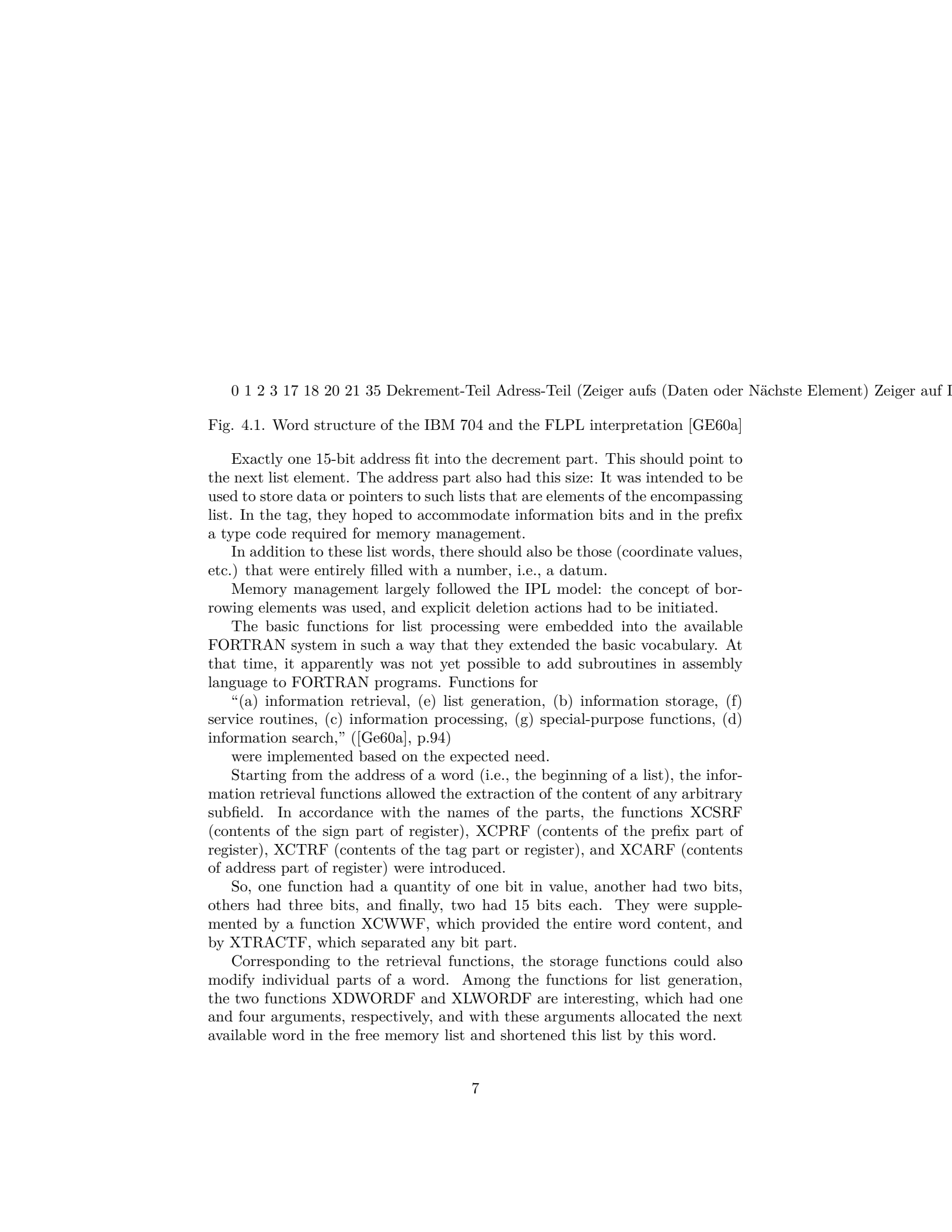

Fig. 4.1. Word structure of the IBM 704 and the FLPL interpretation [GE60a]

Exactly one 15-bit address fit into the decrement part. This should point to the next list element. The address part also had this size: It was intended to be used to store data or pointers to such lists that are elements of the encompassing list. In the tag, they hoped to accommodate information bits and in the prefix a type code required for memory management.

In addition to these list words, there should also be those (coordinate values, etc.) that were entirely filled with a number, i.e., a datum.

Memory management largely followed the IPL model: the concept of borrowing elements was used, and explicit deletion actions had to be initiated.

The basic functions for list processing were embedded into the available FORTRAN system in such a way that they extended the basic vocabulary. At that time, it apparently was not yet possible to add subroutines in assembly language to FORTRAN programs. Functions for

"(a) information retrieval, (e) list generation, (b) information storage, (f) service routines, (c) information processing, (g) special-purpose functions, (d) information search," ([Ge60a], p.94)

were implemented based on the expected need.

Starting from the address of a word (i.e., the beginning of a list), the information retrieval functions allowed the extraction of the content of any arbitrary subfield. In accordance with the names of the parts, the functions XCSRF (contents of the sign part of register), XCPRF (contents of the prefix part of register), XCTRF (contents of the tag part or register), and XCARF (contents of address part of register) were introduced.

So, one function had a quantity of one bit in value, another had two bits, others had three bits, and finally, two had 15 bits each. They were supplemented by a function XCWWF, which provided the entire word content, and by XTRACTF, which separated any bit part.

Corresponding to the retrieval functions, the storage functions could also modify individual parts of a word. Among the functions for list generation, the two functions XDWORDF and XLWORDF are interesting, which had one and four arguments, respectively, and with these arguments allocated the next available word in the free memory list and shortened this list by this word.

McCarthy was not clear about the value of the list generation functions, and in fact, he did not recognize them as functions at all.

A significant difficulty in capturing and evaluating the created "functions" was the fact that one was operating in the FORTRAN concept system, where practically only numbers existed as data structures. How should one understand one of the selector functions as a real function that maps numbers to numbers? "The predominant characteristic of most of our list-processing functions is that they are not functions in the usual sense of the word at all. For many of them, the value of the function depends not only on their arguments but also on the specific internal state of the computer at the time when it is evaluated. In addition, some of the functions not only have numerical values but also produce changes in the internal state of the computer when evaluated..." [GE60a][p.93].

Apparently, this uncertainty led to prefixing an X to the list processing functions as the first letter.[2] McCarthy only grasped the peculiarity of the list processing functions when he understood them as functions over the data structure of S-expressions.

In contrast to this theoretical uncertainty, it gradually became apparent that the functions could be nested and used individually flawlessly like proper functions over numbers. Gelernter and C.L. Gerberich, one of the programmers, also found out that the functions XDWORDF and XWORDF would need to have a 15-bit quantity in the form of the address of the newly constructed list element as their value to be well-suited for nesting.

The IBM group gradually became independent and eventually determined its further direction independently of McCarthy. The various functions added to FORTRAN eventually became a proper system, which they called FLPL (Fortran List Processing Language) [HAN60]. This was later used and distributed independently of the geometric theorem prover.

The work on FLPL was continued by Gelernter, Gerberich, and later by J.R. Hansen. The first successful proof of a geometric theorem was carried out in the spring of 1959; initial results in realizing Minsky's idea were presented at the first IFIP conference in 1959 in Paris [GE59]. Further results can be studied in [GE60a, GE60b, HAN60].

Letter to Morse from December 1957...

Inclusion in ACM Ad-Hoc Group in January 1958...

Proposal of this group from May 1958...

Letter to Perlis and Turanski from June 1958...

In the summer of 1958, McCarthy was invited for a working stay at IBM. By then, he had been working on a chess program that relied entirely on the means of normal FORTRAN. Since he didn't like the conditional statement contained in FORTRAN, the so-called arithmetic IF, with its restriction to comparisons with 0 and the machine-level-like three-part control flow, he used a specially invented function for conditions. This function had three arguments and delivered the value of the second or third when the first argument was 0 or 1,

[2] Actually, Fortran required a function name to begin with X "if and only if the value of the function is to be fixed point" [FOR56]. PMcJ

respectively. Of course, this function enabled some programmatic improvement, but it was still unsightly in its condition on the FORTRAN mechanism. Namely, all three arguments were calculated before the function worked, while in reality, only two would be needed—the condition and the selected sequence.

Since McCarthy had some time this year—he had a research fellowship from the Sloan Foundation—he thought intensively about the problem and developed the idea of conditional expressions. An article on this topic was sent to Communications ACM, which was, however, transformed into a "Letter to the Editor" [MCC59a].

The peculiarity of this invention is the nestability of conditional expressions and the restriction of computation to the parts that belong to the conditions or selected sequences. Compared to both the IF statement and an IF function, this results in an increase in efficiency and expressive power, leading to entirely novel application possibilities.

McCarthy felt the significance of his idea and attempted to demonstrate its usefulness to the ALGOL committee but was unsuccessful [MCC74]. "I did a lot of propaganda for the inclusion of conditional expressions in ALGOL... and in 1958, I lost because the idea was too unusual, and I didn't explain it well enough; so I had no success in getting this thing into ALGOL in 1958." Only in 1960 could McCarthy persuade the ALGOL committee, of which he was now a member. However, a different syntax was preferred, proposed by J. Backus, as they wanted everything not expressed by English keywords to correspond to mathematical notation.

**** IN BOOK, page 65: So, McCarthy spent the summer of 1958 at IBM. Here, he dealt with the problem of symbolic differentiation of algebraic expressions, showing that the ideas developed also went beyond the still emerging FLPL.

During this time, the connection between conditional expressions and recursive functions emerged because "the idea of differentiation is obviously recursive" [MCC78a]. Another significant discovery concerned the usability of functional arguments. If one wants to differentiate sums with many summands, the differential operator must be applied to each of them. McCarthy found it very simple and clear to introduce a function, maplist, which applies a given functional argument to all elements of a list and combines all the results into a new list.

However, the question arose of how to notate functional arguments.

With these new means of expression, the differentiation program in FLPL was not implementable. In general, FORTRAN as the language base was no longer sufficient, but the IBM team was not willing to engage in such a drastic change. Moreover, they found the utility of the innovations highly questionable [MCC78a].

A new language seemed necessary, and it was fortunate for McCarthy that he got a material basis to build on: in the fall of 1958, the MIT Artificial Intelligence Project was launched. McCarthy became an Assistant Professor in the Department of Electrical Engineering (his field was communication science), and Minsky became an Assistant Professor in the Department of Mathematics.

They received support from the Research Laboratory of Electronics (R.L.E.) in the form of two programmers, a secretary, a typewriter, and six students to supervise.

The status of McCarthy's ideas about his new language is reflected in the 1st memo he presumably wrote at MIT in early September 1958. It still shows uncertainty about the list processing functions, the use of 3-bit quantities, IPL-like memory management, conditional expressions, declarative parts (agreements of P-lists), and functional arguments without using λ [MCC58a].

The work is the most important document before the first draft of the "Recursive functions...". In it, McCarthy dealt with the application areas of the language, the most important language elements, and some implementation details. Additionally, he addressed the peculiarities of the new "algebraic language for manipulating symbolic expressions," as the title of the memo suggests.

The application areas targeted by McCarthy are subfields of symbol manipulation: manipulation of sentences in formal languages—important for the Advice Taker Project—simplification and differentiation of algebraic expressions, and compiler programming for the language itself. He emphasized the suitability of the language for formulating heuristic programs because it was very suitable for representing "trees of alternative actions."

McCarthy introduced the language by discussing or specifying the basic data structures, types of statements, and some example programs.

As data structures, the following were envisaged: integers (as indices or addresses), whole words, which might be interpreted as floating-point numbers, truth values, so-called "locational quantities," i.e., addresses in the program itself, such as jump labels, etc., and finally, "functional quantities" to understand functional arguments as data.

The omission of the data structure "list" and instead the sole use of address numbers once again points to the theoretical problems of speaking about functions of lists in FORTRAN. Incidentally, even LISP 1 [MCC60b] had floating-point arithmetic but not for integers! In connection with the basic functions, we will address this problem once again. At this point, McCarthy initially justified the omission of lists: "This (the omission, the author.) is, although a number of interesting and useful operations on entire lists have been defined because most of the calculations we have actually performed up to now could not be described using these operations..." ([MCC58a], p.5). This means that procedures that separated an element from a list or determined an associated number (such as the length) were not seen as realizations of real functions. It can be assumed that not only the FORTRAN conceptual world was responsible for this but also the contemporary state of the implementation of list structures obscured the view. Not for nothing was so much space repeatedly given to this point.

Although lists were not considered as data structures of the language, they could still be read in. The external notation was:

$$(e_1, e_2, e_3, \ldots, e_n),$$

where the e_i are sublists or symbols. In addition to lists, numbers, symbols,

floating-point numbers, and text should be readable. There is no information about how to prepare the text for such input.

The algebraic language should, like FORTRAN, allow a series of statements. In addition to the arithmetic (FORTRAN terminology) or replacement statement, the following should be available: a go statement, a set statement, subroutine calls, declarative statements, a compound statement, suitable instructions for iteration, and finally, elements for defining subroutines.

Assignment (arithmetic statement) should probably be the most important statement, as in FORTRAN. On the right side, arbitrary expressions, composed of simple function expressions through nesting, logical conjunctions, and conditional expressions, were allowed, while on the left side, variables, indexed variables (whose use was still somewhat uncertain), and expressions through which a certain part of a word was selected could be placed. For example, an assignment like

```
car(i) = cdr(j)
```

should be possible.

In the jump statement

```
go(e)
```

go(e)

the argument expression should be computable (of type Locational Quantity).

By the assignment set

```
go(e)
```

a set of assignments should be executed simultaneously. The instruction builds an array A with dimensions 1, ..., m, which receives the values of the q_i in a straightforward manner. These values can then be reused later as $A(i)$.

The subroutine call should be noted without the CALL keyword required in FORTRAN.

Through declarative sentences, P-list structures should be agreed upon. Their general form should be

I declare (dcl_1, dcl_2, ..., dcl_n)

where individual declarations should correspond to two different patterns: Through

(a: p_1, p_2, ..., p_n)

the values of expressions p_1 to p_1 should be successively placed into the P-list for the symbol corresponding to a. ("Every symbol in the program has such a property list, which is used and extended by the compiler but is also made available to the object program.") Through

$$(a_1,\ a_2,\ \ldots,\ a_n,\ p)$$

it is achieved that the expression p is placed into the P-list for the symbols corresponding to the a_i. The last comma is likely meant to be a semicolon.

However, there is still no mention of indicators or P-list access functions anywhere.

Composite statements should consist of a sequence of bracketed simple statements. "The symbols to be used for these 'vertical brackets' have not yet been determined." In the example programs, McCarthy uses the brackets / and \. If an entire function consists of an expression, the statement brackets are not needed.

The form of the iteration statements was not yet established. "... it should include the FORTRAN type, as well as a DO over an explicitly given list..." [MCC58a][p.9].

Also, the subroutine definition should largely resemble the corresponding FORTRAN language form. Additionally, functions and Locational Quantities (and expressions from them) should appear as parameters; they should be separately compilable, and symbolic references to other variables in subroutines should be guaranteed.

The functions intended for list manipulation were divided into those for basic operations regarding whole list structures and the basic functions for individual words (basic single word functions).

```
/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
```

a) The function copy(J)

```
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))
```

b) The function equal(L1, L2)

Fig. 4.2. Notation in Lisp, September 1958 [MCC58a]

The former were exemplified by the functions eralis(j), copy(j), search(l, j, p_1, p_2, p_3, m), maplist(l, j, f(j)), and list(j_1, ..., j_n). In this context, eralis deleted an entire list structure and added it to the free storage list, copy copied a list structure, search looked in list l for the first element satisfying the condition p_1, maplist formed a new list whose elements were created from the original list l by applying the function f(j) to each element j, and list formed a list of n elements. Additionally, the function equal was considered.

McCarthy did not consider the basic functions for individual words as realizations of true mathematical functions. As justification, he provides nearly

the same properties of selector functions that had already been emphasized by Gelernter in connection with FLPL: memory dependency and side effects.

In contrast to FLPL, McCarthy came up with two sets of access functions: those whose argument should be an actual machine word and those whose argument should be the address of such a word. The basis for naming his functions is a slightly modified word breakdown.

Each of the parts was assigned a characteristic letter: w (to the word as a whole), p (to the prefix), i (to the indicator), s (to the sign), d (to the decrement), t (to the tag), and a (to the address part).

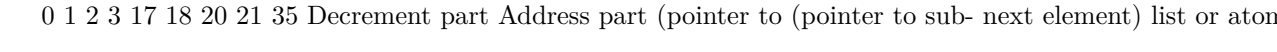

Fig. 3. Word structure of the IBM 704 and the LISP interpretation [MCC58a]

The first class of basic functions included the extraction functions pre, int, sgn, dec, tag, and add. In addition, there was a function that only provided a specific bit: bit, and one that extracted any segment from the word: seg. To reassemble a word, there were comb4(t, a, p, d) and comb5(t, a, s, i, d), as well as a bitwise function choice.

In the second class, the functions cwr, cpr, csr, cir, edr, ctr, and ear were combined. Their argument was a 15-bit address. Naturally, they were related to the functions of the first class. So, the equation:

```
add(ewr(j)) = car(j).
```

Corresponded to the functions bit and seg, the functions cbr and csgr.

The functions that provided 15-bit quantities, i.e., addresses, could be on the left side of an assignment, describing the target of an assignment. The same effect could be achieved by calling the storage functions stwr, stpr, stsr, stir, stdr, sttr, and star. Each of these functions had two arguments: the address (as a number) of the word to be stored and the data to be stored.

The construction functions in this class worked similarly to the FLPL functions XDWORDF and XLWORDF: They removed one or two words from the free storage list, shortened it accordingly, and filled the word(s) with their arguments: consw had a whole word as an argument and filled the free space from the free storage list with it; consel had only two arguments, occupying the address or decrement part with them, filling all other fields with 0; consls worked almost the same way but filled the indicator field with 2 (indicating a

sublist); consfl finally took two words from the free storage list; the first one was completely filled with the first argument of consf, the second one had a pointer in the address part to this first word, and the second argument in the decrement. The address of this second word, where the indicator was set, was the result.

The class was completed by the erase deletion function and the pointer movement functions point, mova, movd, and movup.

We do not want to analyze the ideas for implementation, which were largely based on FLPF, as too much was unclear at that time. Much more interesting, however, is the use of functional arguments, as lambda notation was not yet used.

Already when mentioning the function maplist, it was noticed that it should have three arguments: Since an expression should be the third argument, the local variable had to be made known in the second argument. This can be seen nicely in the example program. By the way, it can be recognized that the atom test should be done by the question: "Is the indicator = 1?"

```
The Program is: 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",
mapist(cdr(J),L,(L = K → diff(L), L = K → copy (L))))))) return
```

Fig. 4. The function diff for differentiation [MCC58a][App. I]

Undoubtedly, it is difficult to imagine that this program should be executable. But McCarthy probably felt the same way. He may have tried many other variants.

So most likely, the discussion with this program and its implementation led to the inclusion of lambda notation in LISP. Whether the idea came from McCarthy alone, or whether Rochester, Minsky, or Shannon, who were professors at MIT at the time and closely connected with McCarthy's work, gave a hint here, is not known so far.

It is interesting that McCarthy's arguments to change the definition of maplist so that it only has two arguments and a function appears as the second one are mainly derived from the implementation: McCarthy seems to have tried to write an assembler program for maplist [MCC58b]. This had two weaknesses: On the one hand, the variable that appeared as the 2nd argument was needed too often in the expression (3rd argument). This could not be managed. On the other hand, the variable could be integrated very poorly into the call sequence for maplist: What should then be passed as an argument?

By imagining that the function appearing as an argument would be embodied as an instruction for calling this function in maplist, McCarthy was able to achieve a satisfactory definition and implementation. The definition equation written in LISP now was [MCC58b][p.2]:
maplist(L,f) = (L = 0 → 0,1 → consls(f(L), maplist(cdr(L), f))).

With this new form of the functional, the differentiation function could be rewritten:

```
   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))))))))
```

Fig. 5. The improved function diff with λ-notation[MCC58b]

One can see that further improvements have been made: approaches to distinguish variables from constants are evident. In addition, it can be seen that 1 is used as the truth value true, consequently, 0 is surely false. From the above maplist definition, it can also be read that 0 denotes the end of the list: However, one 0 was a bit size, and the other 0 was a 15-digit address. For convenience in implementation, both were quickly mixed up. The name of the later LISP basic function NULL also resulted from these implementation details.

Finally, attention should be drawn to a small detail: In the discussion of the peculiarities of this language for symbol manipulation, where McCarthy discusses algebraic notation and its advantages, recursion and its implementation using push-down lists (according to Newell, Shaw, and Simon: real lists!), conditional expressions, and functional parameters, it is also briefly mentioned that expressions can be conveniently represented by lists. This is not limited to data expressions [MCC58a][p.2]. Later (p.11), he shows that his functional expression $f(e_1, ..., e_n)$ is represented by the sentence `(f,` e_1`, ...,` e_n`)`, and notes that it can thus be realized as a list structure. Together with the statement that the compiler should also be written in the language itself, it can be inferred that initial ideas for representing programs as list data had already been developed.

As the programming language, now somewhat concretized, was necessary for the realization of the ideas of both project leaders - McCarthy had put his ideas into a program that was supposed to demonstrate artificial intelligence in his work on the Advice Taker [MCC58c] - implementation began. However, no one knew exactly how to write a compiler, and the example of FORTRAN was rather daunting, as it had required 30 man-years, which was an unimaginably large effort for that time.

To gain experience, various functions were manually translated into assembly language. Conventions for subroutine calls, stack storage work, and memory management were to be developed. This work was mainly carried out by the programmers K. Maling and S. B. Russell. But also the students, among whom were D.G. Bobrow, D.C. Luckham, D.M.R. Park, and R.K. Brayton as well as L. Hodes, L. Kleinrock, and J.R. Slagle, had a certain share.

In the manual translation of the input and output functions, a practical external standard notation for symbolic expressions was created, using round brackets (parentheses) as list brackets[3] and a prefix notation in which all mathematical operators were treated in a uniform syntactic way. Thus, $a + 2b + c^2$ becomes (PLUS A (TIMES 2 B) (EXP C 2)). This notation was later called "Cambridge Polish" to honor J. Lukasiewicz and W. V. Quine.

[3] McCarthy is no longer sure today whether this specific decision was made earlier [MCC74]

The external notation was relatively poor in special characters due to the limited character set of the IBM 026 key punch.

However, the programming language, which got its name LISP around this time, had a larger character set since it never had to be punched, but was only used for "paper programs." Essentially, it still corresponded to FORTRAN, only conditional expressions and list processing functions and recursive functions could be defined as well. To mention list constants in the program, square brackets (brackets) were used when one wanted to combine arguments of a function. The later so-called M language [MCC62c] is likely to largely correspond to LISP at the end of 1958, although the term prog was missing. Nothing is known about declaration rules.

```
diff[s;v] = [atom[s] -> [eq[s;v] -> 1.0; T -> 0.0]; eq[car[s]; PLUS]
-> cons[PLUS; maplist[cdr[s]; λ[[si]; diff[car[si]; v]]]]; eq[car[s];
TIMES] -> cons[PLUS; maplist[cdr[s]; λ[[si]; cons[TIMES; maplist[cdr[s];
λ[[sj]; [equal[si;sj] -> diff[car[sj]; v]; T -> car[sj]]]]]]]] a)
pairmap[k; m; farg; z] = prog[[b]; A [null[k] -> return[conc[reverse[b];
z]]]; b := cons[farg[car[k]; car[m]]; b]; k := cdr[k]; m := cdr[m];
go[A] b)
```

Fig. 6. Two examples of the M language with prog.

The most complex problems in translating LISP programs into assembly programs turned out to be the implementation of recursive functions and the deletion of unnecessary list structures.

The organization of recursive functions (and subroutine work in general) was so important because they became the basic concept. One could resort to the example of IPL when solving the problems associated with it. However, the implementation of stack storage as lists seemed unsuitable, and McCarthy decided to use linear sequences of machine words for acceleration. While initially thinking of assigning each function its private stack, this idea was quickly abandoned. The central stack storage remained.

The subroutine entry conventions were now completed, so that when a subroutine was called, the required temporary registers (memory words) were uniformly saved in the stack storage to be available in the program itself. In each of these blocks in the stack, the name of the function, the size of the block, and its origin from memory were later described.

In accordance with the provisions made upon entry, the leaving of subprograms was also standardized.

These works were indeed pioneering achievements because, although the stack principle was known, it had not yet been utilized for systematic implementation. E.W. Dijkstra's publication on his ALGOL implementation did not occur until 1960 [DI60].

Deleting unnecessary list structures proved to be a difficult problem. IPL had introduced a very complex system for this purpose. Lists could generally only be accessed through a specific variable. However, they could be borrowed for use. If the programmer wanted to return such a list to the memory, he had to be sure that it was no longer borrowed anywhere.

There was an explicit deletion operator. External devices seem to have been used for the deletion process itself [MCC74]. To delete only the parts that were no longer needed during this process, availability bits were introduced. Only the parts that were truly free and available at the time of deletion were released for deletion. Determining the availability of a substructure became an important and complicated task because this availability constantly changed, and the "borrowed" status was not easy to oversee.

In IPL programs, an availability error occurred frequently; it was referred to as a space thief when important list structures were deleted. Often, such a program continued to run with destroyed list cells for a long time before terminating with completely corrupted error messages.

At any given time, an idea from G.E. Collins [COL60, COL63][4] suggested determining availability through reference counters. These counters were assigned to every word that was part of a list structure and contained the number of references made to that word. If one wanted to delete substructures, only those with a reference count of 0 were actually eliminated. If a substructure could be deleted, all reference counters referred to by that substructure were reduced by 1.

Pursuing this idea for LISP proved to be unfavorable on the IBM 704, as this machine had six free bits in each word, but they were separate.

The new idea was that the programmer should no longer be concerned with deletion. This was initially the simplest solution, as there were no memory problems yet, and the problem could be postponed.

It was only in 1959 that the concept of dynamic memory management and its core, the regenerator (reclaimer), was developed (probably by McCarthy himself). Only when the memory space was exhausted did a process begin, in which all reachable words were determined, and the unreachable ones were placed in the free memory list. Programmers referred to this process as garbage collection, a term that initially seemed unusable in publications due to its vulgarity (it was then replaced by "reclaimer" by reviewers) but has since become widely accepted.

In the context of garbage collection, it was necessary to determine which list structures should be reachable. Since all those that had been temporarily stored in the stack memory had to be counted anyway, care had to be taken to ensure that other information stored in the stack could not be interpreted as pointers in the list memory. Thus, integers and floating-point numbers had to be converted to a range of constants and referenced with pointers. However, initially, numbers seem to have been represented as lists of digits.

Garbage collection was thus invented as a program, as was its subordination as a subroutine of CONS. Later, efforts were made to provide users with the results of garbage collection as memory statistics.

In this way, a program system gradually emerged that was expandable through hand-translated LISP functions. Of the programs embedded in this system, only one program system for handling electrical networks is known,

[4]It is not clear whether this idea was already developed in 1958.

which, according to J. Moses [MOS71], included the first simplification program: [GOB59, ED59], and also Maling's differentiation program [MAL59].

Some of the provisions proved to be unfavorable for the future. This includes the idea, technically convenient for implementation, of ending lists with 0 and identifying the atom used as a truth value for "false" with it (initially, even 0 itself was used). This atom was called NIL (or later NIL), but the question about NIL is handled with the function NULL. "Apart from encouraging pornographic programming, assigning a special meaning to address 0 has caused difficulties in all subsequent implementations." [MCC78a].

However, a definition of the LISP programming language did not exist at that time because a compiler was lacking, which would have enforced a precisely defined syntax. Rough provisions were sufficient for translating paper programs into assembler form, and a comment sufficed if necessary. In this situation, an article by McCarthy, which he began writing in early 1959 [MCC59b] and which appeared in Communications ACM in early 1960 [MCC60c], was to become crucial. McCarthy wanted to present his view that his language, as it was at the time, was just as suitable for the theory of computability as a means of description as the Turing machine. Of course, it was also more practical because essential things could be executed much faster ("... one could really prove that it is better (neater)... one could make less complicated proofs...") [MCC74].

To demonstrate that LISP can not only describe computable functions much more easily but also achieve the same functionality, McCarthy chose the task of writing a general (universal) LISP function and showing that it is shorter and more understandable than the description of a universal Turing machine. A universal TURING machine is characterized by the fact that it receives the description of another Turing machine and data for it on its input tape and can then imitate it. Therefore, the universal LISP function had to accept the description of LISP functions as data and demonstrate corresponding performance, i.e., be able to execute the work of the relevant functions.

For this purpose, a notation had to be found in which all LISP expressions and LISP functions could be described as symbolic expressions. This is not so simple because LISP functions can contain constant symbolic expressions that do not differ in any way from the descriptions to be generated. By introducing quoting for such constant symbolic expressions, this problem was overcome, and McCarthy arrived at the well-known translation rules [MCC60c][p. 186f.]:

1. If E is an S-expression, then (QUOTE, E) is the translation result.

2. Variable and function names represented by sequences of lowercase letters are translated into the corresponding sequences with uppercase letters.

3. A form f`(`e_1`;` e_2`;` `...`; e_n`)` is translated into the expression `(`f^+`,` e_1^+`,` e_2^+`,` `...`, e_n^+`)` (where the index + indicates the result of the translation process on subexpressions).

4. `[`$p_1 \longrightarrow e_1$`;` `...`; $p_n \longrightarrow e_n$`]` is translated into `(COND (`p^+1`,` e^+1`)` ... `(`p_n^+`,` $e^+ - 1$`))`.

5. λ[[x_1, ..., x_n]; e] is translated into (LAMBDA (x^+1, x^+2, ..., x_n^+) e^+).

6. label[a;e] is translated into (LABEL, a^+, e^+).

The label operator seemed necessary to be able to denote recursive functions as well. When there were believed to be difficulties with the λ-calculus in this regard, Rochester proposed this change. However, it could have been done without it, namely with a construction that corresponds to Church's paradoxical combinator Y.

Of course, the significance of the decision to write data and programs in the same notation was not clear from the beginning. The function EVAL, equivalent to the universal Turing machine[5], was initially intended only as a theoretical vehicle to be able to prove the translation process as well.

However, the writing of the article and the formulation of the expression-evaluating function had a number of unexpected consequences. These were mainly triggered by Russell's random decision to translate the EVAL function into an assembly program. "Steve Russell said, 'Give it here, why shouldn't I program this EVAL...?' But I replied to him, 'Ho ho, you are confusing theory and practice because this EVAL is meant for reading, not for calculation...' But he went ahead and did it..." [MCC74].

At that time, when McCarthy was working on his article, the compiler project had not yet really begun. In the situation without a compiler, the implemented EVAL function served as a makeshift interpreter[MCC76]. However, since it offered the possibility to connect LISP programs to the running system with a simple transformation, it was also used.

The use led to the need to strictly transform programs built inconsistently from expressions and statements into nested expressions. Now it became necessary to introduce the "function" PROG to save a certain part of the familiar FORTRAN. LISP is only an expression language since the implemented EVAL function existed.

The interpreter was also not taken very seriously at that time. Some of the early implementations of LISP (at least on the Q-32) completely dispensed with interpretation without having to deviate from LISP. Only when LISP was embedded in a time-sharing system did all the advantages of processing through a read-evaluate-print loop become apparent.

"The unexpected existence of an interpreter led to the freezing of the form of the language" writes McCarthy [MCC78a], characterizing the undesired consequences of the EVAL implementation, "and some of the decisions that had been made somewhat casually for the publication of 'recursive functions...' later proved to be unfortunate. These include the COND notation for conditional expressions, which leads to an unnecessary number of parentheses..."

"One reason for the initial willingness to live with the disadvantages of the internal form of LISP was that we still hoped to write programs as M-expressions.

[5]D. Park claims that in November 1958, when he joined McCarthy's group, APPYL-EVAL was already working. It may be that the working system was more the totality of hand-translated functions at that time.

The project to precisely define M-expressions and to compile them or at least translate them into S-expressions had neither been completed nor explicitly abandoned. It had only been postponed to an indefinite future, but then a new generation of programmers appeared who preferred the internal notation of any FORTRAN- or ALGOL-like language..."[MCC78a].

However, the path from publication to implementation was not quite direct. In the translation and testing process, Russell and Maling must have eliminated a number of errors. To the present-day reader with some knowledge of LISP, these errors naturally stand out. It is somewhat curious that C.R. Jordan wrote a critical publication in 1973 [JO73] about the errors in this first EVAL function.

In addition to notational errors (use of undefined functions and unclear characters (e.g., = - clearer in [MCC60c] than in [MCC59d]), the undesirable property can be found that with each determination of the function name in the association list (where the local variable meanings are stored), the arguments are evaluated. That a function could be calculated was completely ignored. These errors were corrected in the later published EVAL-APPLY functions [MCC62c, MCC63e].

The article on the universal LISP function had significant consequences for McCarthy's scientific interests. First, his interest in the theory of recursive functions was significantly increased, and he continued to explore how a usable theory of computability could be built on his LISP calculus. Reflecting on what such a theory should accomplish led him to the problem of proving properties of programs, etc., as he explained in his fundamental work in 1963 [MCC63a]. He expressed his thoughts on this topic in early 1961 [MCC61b].

On the other hand, as he later realized, he had not only written a universal LISP function but also simultaneously described the semantics of the (still unknown to him) LISP language. Dijkstra's judgment [DI72] about the LISP definition "as a curious mixture of what the language means and how the mechanism works" hits the mark, even though it misses the point: the definition was not intended!

It is certain that McCarthy later proceeded from his approach to describing LISP to make proposals for the definition of the semantics of programming languages [MCC64], and these proposals were immensely influential. The Vienna Definition Language is a descendant of his ideas.

In summary, quite suddenly and unintentionally, the implementation of EVAL created a second language form alongside the existing LISP language. Initially, the language was called S-language after symbolic expressions (the LISP data). The older form was named M-language because discussions about working in the S-language (e.g., the evaluation mechanism) could be conducted in it (M for meta). Although the M-language was soon only used for designing programs (if at all) and a translation program from M-language to S-language was not even written (this may have been due to the unavailability of a device with the required character set), it never disappeared completely. McCarthy himself apparently always preferred the M-language. J.R. Allen's book on LISP [ALL78] uses only the M-language and calls the S-language a "representation form." E. Sandewall [SAN75b] emphasizes the independence of the external no-

tation from the internal representation, which reflects the S-expressions. However, LISP today is identical to the S-language of that time.

With the completion of the translator for the EVAL function and its improvement, the first LISP system had progressed so far that it was directly usable. By inserting the functions APPLY, EVAL, EVCON, and EVLIS [MCC60c][p.189] into the existing programming system, an interpreter-oriented system was created, the advantages of which became apparent through McCarthy's simultaneous work on time-sharing systems.

By the way, a beautiful mistake [ALL78] had crept into the system quite early on, which significantly influenced the form of the language: the print program forgot to print the commas or delivered a non-printable character. In any case, it was found that the lists were much more readable without the commas. So, the print program was left as it was, and the notation was changed.

For a long time, the comma occupied a curious position: Every LISP program could also be noted with commas, and LISP systems would usually have read it correctly. But no one used the comma in this way. However, because the comma became a syntax character, it led a true shadow existence: Incorrect usage would have caused syntactic errors, and correct usage was not common! It was, in a way, a non-character. Only the idea of describing structures to be built with a backquote (‘) in a goal-oriented way and indicating variable components to be incorporated has awakened the comma from its slumber.

It is not possible to specify an exact date for the completion of the first LISP system. If we rely on Park's recollection, his early EVAL-APPLY version was already running in November. Gradual extension, improvement, and revision of the system, individual functions, or modification of atomic structures should have resulted in a gradually usable programming system with diminishing constraints.

Students D.J. Edwards, S.H. Goldberg, P. Markstein, and C.S. Rubenstein likely began their work on using symbolic manipulation for the calculation of electrical circuits early in 1959. Unfortunately, the progress report from early January 1959 [MCC59b] contains no indications.

By April, the state of affairs was already well developed. A whole series of programs explicitly mentioned were written in LISP. However, it seems that the assembly language was still used alongside, probably for the language program being developed by Shannon, McCarthy, Abrahams, and Kleinrock. “The current status of the system can be summarized as follows:

(a) The source language has been developed and is described in several memoranda from the Artificial Intelligence group.

(b) Twenty useful subroutines have been programmed in LISP, hand-translated into SAP (symbolic machine language for the IBM 704 computer) and checked out on the IBM 704. These include routines for reading and printing list structures.

(c) A routine for differentiating elementary functions has been written. A simple version has been checked out, and a more complicated version that

can differentiate any function when given a formula for its gradient is almost checked out.

(d) A universal function apply has been written in LISP, hand-translated, and checked out. Given a symbolic expression for a LISP function and a list of arguments apply computes the result of applying the function to the arguments. It can serve as an inter- preter for the system and is being used to check out programs in the LISP language before translating them to machine language.

(e) Work on a compiler has been started. A draft version has been written in LISP, and is being discussed before it is translated to machine language or checked out with apply.

(f) The LISP programming system will be shown in this report to be based mathe- matically on a way of generating the general recursive functions of symbolic expressions. " ([MCC59c][p.122]).

In the report itself, which represents the draft for the work "Recursive functions..." [MCC60c], McCarthy mentions two compiler versions and garbage collection.

Edwards and Rubenstein completed their work for the Bachelor of Science degree by the summer of 1959; in the fall, Goldberg completed his Master's thesis.

At the ACM annual meeting in early September 1959, McCarthy presented his programming language LISP to the public for the first time [MCC59g]. Nothing is known about the reception. A similar demonstration of the system, which could be interactively used via a typewriter (the Flexowriter system by Luckham and Edwards), is humorously described by McCarthy in [MCC78a].

Edwards remained at MIT after completing his degree and took Maling's place from September 1959, as Maling is no longer mentioned as a group member from 1960 onwards. During this time, McCarthy started working on a manual for LISP but did not complete it [MCC59f].

The period until early 1960, when the article "Recursive functions..." [MCC60cc] and the LISP 1 Manual [MCC60b] were published almost simultaneously, and work on the IBM 709 began, was largely occupied with the work on the compiler and solving the problem with free variables.

Brayton, supported by Maling and Park, finally had the compiler working largely free of errors. Accelerations of around 60 times or figures such as: runtime—interpretative 30 minutes, compiled 10 seconds, are typically indicated. Roughly characterized, the compiler can be described as both very similar to later programs of the same kind (after all, it was the prototype!) and, on the other hand, different from them.

On the one hand, the compiler already generated a list of symbolic code—in "LISP Assemble" (LAP). Subsequently, this program was assembled and added to the system as new machine functions. Many of the program parts required for this were either directly adopted or modified by rewriting in later compilers.

On the other hand, the produced code was, in its nature, very different from the code generated by later compilers: Spaces for local variables were generated by GENSYM and appended at the end of the program. Recursive functions had to explicitly move these local variables into the stack before they could start their work. Since the command format did not consist of instructions and operands as it would later, but rather an element was reserved for labels, no atoms appeared within the LAP code level.

To achieve effective programs, the functions CAR, CDR, CONS, and NULL were translated "openly"[6]. Naturally, this applied to all special forms (PROG, GO, RETURN, COND).

```
COMPILE
(((LABEL, MEMLIS,(LAMBDA, (X, Y), (PROG, (M1),(SETQ, M1,Y), M2,
(COND,((NULL, M1), (RETURN, F)), ((EQUAL, (CAR, M1), X),
(RETURN, T))), (SETQ, M1, (CDR, M1)), (GO, M2))))))
(MEMLIS, BSS, 0)
( ,SXD, G0001, 4)            save index Register 4
( , STO, G0002)              store x
( , STQ, G0003)              store y
( , CLA, G0003)
( , STO, G0007)              set m1 = y
(GOOO6, BSS, 0)              location M2
( , CLA, G0007)
( , TNZ, G0008)              to G0008 if null[M1] = F, or
( , CLA, $ZERO)              return with AC = F if null M1 = T
( , TRA, G0005)
(GOOO8, BSS, 0)
( , LDX, G0007, 4)
( , CLA, 0, 4)
( , PAX, 0, 4)               car[M1]
( , SXD, G0011, 4)
( , LDQ, G0002)              x
( , CLA, G0011)              car[M1]
( , TSX, EQUAL, 4)
( , STO, G00010)             result of equal test (note that the
( , CLA, G00010)             compiler stores everything it computes)
( , TZE, G0009)              transfer if car[M1] =x
( , CLA, $ONE)               otherwise return with AC = T
( , TRA, G0005)
(G0009, BSS, 0)
( , LDX, G0007, 4)
( , CLA, 0, 4)
( , PDX, 0, 4)               cdr[M1]
( , SXD, G0007, 4)           set M1 = cdr[M1]
( , TRA, G0006)
```

[6]Possibly also ATOM was added

```
(G0005, BSS, 0)
( , STO, G0004)              answer (true (one) or false (zero))
( , CLA, G0004)
( , LXD, G0001, 4)           restore original index Register 4
( , TRA, 1, 4)               return
END OF APPLY, VALUE IS ...
(MEMLIS)
```

Fig. 7. Compilation of MEMLIS in LISP 1 ([MCC60c][pp. 58, 59])]

Also, the variable problem was solved during this time. One of the first major users of the LISP system, Slagle, encountered inconsistencies in the course of programming for his integration program. He worked with great intensity, as McCarthy [MCC76] recalls, and sought efficient solutions for the sub-functions. For this, he used functional arguments in one case and the now-built-in ability to note variables and arbitrary expressions in functional positions instead of function names ("... this is logically abnormal, and I rejected this usage. However, I believe some still used it" [MCC76])

Slagle tried to write a function that should test whether a given predicate would be satisfied by all parts of an S-expression. He wrote it "... in a nice recursive way with functions of functions as arguments, but it didn't work. So, of course, he complained. I never quite understood this complaint and just said, 'Oh, there must be a hidden error somewhere.' Steve Russell and Dan Edwards claimed that there were fundamental difficulties with this, but I didn't believe it and asked them to find the error." [MCC74]

Someone, at least with the involvement of Russell and Edwards (Park points to P. Fischer [MCC78a]), then identified the cause of this problem as conflicts between the values of global or free variables in the function definition and their values when the function is called. These conflicts do not normally occur because all functions are defined at a base level. However, since λ expressions also represent functions, the use of global variables in such λ expressions can be problematic when they are transported over a functional argument or an evaluation result from the environment they were originally fitted into. Similar problems occurred shortly afterward in ALGOL 60 compilers (most recent error).

To eliminate the difficulties, FUNARG was introduced (this solution practically corresponds to the introduction of dynamic and static chains in ALGOL 60 [BERR70]).

"They tried to explain it to me, but I was distracted or at least distracted and not attentive, so I really didn't understand what the FUNARG thing did at first, but only a few years later. But then it became known that the same kind of difficulties also arose in ALGOL 60, and at that time, the LISP solution to it, which Steve Russell and Dan Edwards had simply cooked up to eliminate the error, was a more comprehensive solution to this problem than the one that existed in ALGOL compilers at the same time." [MCC74].

However, it has since been shown that the solution to the problem was rather half-hearted. At that time, the problem was underestimated, and a

bridge was built around the "error" (a patch). The solution was correct, but one would have had to intervene in the language and correct the notation and treatment[7] of functions (especially those described with λ). It was believed that the error situation would rarely occur because functions are usually named. The fact that several explanations of Slagle's function use the problem for a practical example already points to this misunderstanding. Similarly, these demonstrations are indicative (although the crucial point is correctly worked out each time); however, errors indicate a lack of understanding of what such a function might be useful for [SAU64a, MCC78a].

For many years, FUNARG was only used in exceptional cases. The use of unnamed functions was at best considered a notation trick to program more cleanly in a function-oriented way. Accordingly, implementors quickly separated themselves from the exotic-looking language element when it stood in the way of implementation decisions. Over time, programmers came to the opinion that FUNARG was superfluous—and thus the static semantics of program variables—and instead, LISP was based on dynamic semantics.

The more difficult-to-understand semantics were accepted because they brought some advantages: Instead of noting functions that use common variables in tree structures and thus making the scopes of these variables explicit, it was preferable to work with linear files of functions. This most practiced programming style is more oriented towards independent functions (at most, global variables are considered "harmful"). One defines a set of functions that stand side by side on an equal footing. Subordination is established by passing control at runtime; variable scopes are kept as separate and local as possible; truly global variables are rarely used.

Additionally, it should be noted that this interpreter-oriented view associates program variables more closely with LISP symbols—the names of variables remain important. A term generated during program execution (or a subroutine) can be processed in the course of its evaluation with reference to dynamically existing variable assignments.

In a certain sense, this programming style is practical and manageable, and it can be reconciled with many demands that modern programming methodology imposes. The problem was only that in the vast majority of LISP systems, there was a difference between the compiler and the interpreter: While the interpreter worked with dynamic semantics, compilers usually referred to the static semantics. Specifically, the work with program variables was changed from accessing the value cell to accessing the argument stack.

Around 1970, J. Weizenbaum and, based on him, Moses argued that the problem was actually due to the incorrect implementation of the λ calculus. Initially highlighted by Sandewall and by Bobrow and B.L. Wegbreit...

One of the few systems that tried to achieve consistency between the interpreter and compiler was InterLISP. However, an attempt was made to realize dynamic semantics. Still, the programmer could use an elaborate implementation that realizes FUNARG through additional notational effort. It was only

[7] which did not correspond at all to McCarthy's translation process!

in the wake of SCHEME that many LISP programmers became aware of the possibilities that correct static semantics offers.

The LISP 1 Programmers Manual (dated March 1, 1960) was written by P.A. Fox, who joined McCarthy's group in September 1959. An interesting system is described in it, consisting of about 90 functions at that time, some of which were highly peculiar and specific (such as the incomprehensible CP1 function in LISP 1.5), while others originated from the work of Edwards and Goldberg. Functions for simplifying, differentiating, multiplying matrices, and other matrix manipulations likely implemented in assembler work survived as system parts.

An essential part of the system was the aforementioned Flexowriter system, based on an idea anticipating time-sharing (time-stealing), enabling interactive work.

Now, there was a system whose development had only slowly progressed due to a lack of concept [MCC76]. Intellectual leadership was entirely in the hands of McCarthy and Minsky; the actual programming work was carried out by Russell and Maling (later Edwards). Furthermore, Hodes, Luckham, and Park were involved in the initial implementation. Of these, only Maling can be attributed to a specific object, the input and output programs. The work of Brayton and Park on the compiler has been mentioned several times.

Regarding the working atmosphere at that time, Abraham [AB76] provides a perspective: "The original system was developed by two employed programmers, Steve Russell and Dan Edwards[8], in addition to a number of graduate students listed in the LISP 1.5 Manual. We all worked in a large room in the same building as the IBM 704. Project meetings, which took place once a week, were also held here. Our room was right next to the workspaces of M. Minsky and J. McCarthy. So, the meeting always began with J. McCarthy entering through the connecting door. The first implementation grew somewhat helter-skelter, and it took quite some time for the efforts to be somewhat coordinated. The machine capabilities were not too good, and LISP jobs took a lot of time. A specific standard run always took a little more than 5 minutes, so Steve and Dan had a standard trick: they ordered the machine for a presence test, which was limited to 5 minutes. In the end, they always managed to persuade the operator to grant them an extra minute, of which, of course, they knew in advance that it would be necessary."

McCarthy also mentioned the lack of organization [MCC76]: "The atmosphere was too loose at the beginning, partly because the project itself had an informal existence (there was never an official start for the MIT Artificial Intelligence Project). There was no written proposal for its start, and all participants were supported (or employed) by the MIT Computer Center and the MIT Research Laboratory of Electronics (R.L.E.). The only real project people were the secretary (M.B. Webber) and the programmers, and our only device was a punched tape punch. If someone lost interest in the work, they simply switched to a new occupation."

[8] the latter hired in September—H.S.

The students were mainly users of the system. The work of Goldberg [GOB59] on the calculation of electrical networks and of Edwards [ED59], who used Goldberg's results, had already been mentioned. The arithmetic simplifier developed by Goldberg was used by other students, such as Rubenstein [RUB59] and Markstein [MCC59c], and further developed in the work of T.P. Hart [HART61]. Moses [MOS71] states that Goldberg's program was still written in assembly language and can be considered the first existing simplification program. It is also known that this simplifier was used in Slagle's integration program [SL61]. The differentiation program by Maling [MAL59] has also been mentioned.

As an additional main activity, the verification of mathematical proofs can be considered, with which McCarthy [MCC61a], Abrahams [AB3,64], as well as Bobrow, Maling, and Park [MCC59c] were involved. In addition, work on automatic proof generation was undertaken; Bobrow and Raphael dealt with natural language processing and simple question-answer systems [BO63b, RA63]. Other projects included the development of a mechanical hand (Bobrow, Minsky, and Shannon), chess programs (Abrahams, Kleinrock, McCarthy, and Shannon), and the Advice Taker (McCarthy, Minsky, Russell [MCC59b, c, 60a]).

From these activities emerged doctoral theses for the students, defended between 1961 and 1963.

It is noticeable that none of the students addressed the theoretical problems raised by McCarthy's work in recursive function theory or the theory of computability. Equally peculiar is that the students listed in the LISP 1 manual did not pursue a Ph.D. related to LISP or using LISP.

Park [PAR78] writes about this: "I think that McCarthy, while superior to us in assessing the importance of the matter, ... , was just as surprised as everyone else that his thing had such a general impact on computer science. LISP made everything look easy. Paradoxically, for those of us looking for research topics, things seemed too easy in a certain sense. McCarthy began to show much of the theoretical significance in his work on the 'Mathematical Theory of Computation'—but he had to do almost all of it alone. The rest of us, especially myself, grasped the message much more slowly. None of the research students listed as co-authors of LISP 1 started a dissertation using computers or with a programming theme. I myself, who was excited about LISP and watched McCarthy's initial work on his theory of computation with unwarranted anger because it seemed too simplistic for such an intellectually ambitious student like me, ended up with a Ph.D. in mathematical logic..."

In early 1960, the implementation of the LISP system for the IBM 709 had begun, as reported by McCarthy in [MCC60c]. In the time that followed, a good number of students left, and others, including Hart and M.I. Levin, joined and participated in the work. At some point, the IBM 709 was replaced by an IBM 7090, and by mid-1962, LISP had undergone so many improvements that a new manual was needed. Levin compiled it. It was completed on August 17, 1962, and was titled the "LISP 1.5 Programmer's Manual."

The differences between LISP 1 and LISP 1.5 can be described as follows:

1. Differences in Data Structures:

    a) LISP 1 allowed only floating-point numbers, not integers. These were structured similarly to atoms and were grouped in a list of floating-point numbers. When evaluated, any occurring floating-point number had to be quoted.

    LISP 1.5 introduced integers and significantly changed the structure of both types of numbers. Many objects of the same value can exist. Numbers can be evaluated without quoting—they are the result themselves.

    b) LISP 1.5 allowed dot notation (A . B).

    c) LISP 1.5 introduced arrays.

    d) LISP 1.5 also changed the atom structure (only APVAL remained as an indicator of the global value).

    e) LISP 1.5 introduced the notation $$x ... x for atoms with special characters.

    f) LISP 1 did not allow cyclic structures.

2. Differences in Functions

    a) LISP 1 allowed functions with only 10 arguments. LISP 1.5 extended this to the number 20.

    b) In LISP 1, only the LIST function could have an indefinite number of arguments. LISP 1.5 systematized this and introduced a whole range of such functions.

3. Changes in the Function Spectrum:

    a) LISP provided only the functions SUM, PRDCT, and EXPT for arithmetic (addition, multiplication, and exponentiation). LISP 1.5 has a broad spectrum (see Section 5.3).

    b) LISP 1.5 introduced input and output functions for character manipulation.

    c) In LISP 1, auxiliary functions were introduced to efficiently process functions of the CAR-CDR family (DESC, MAKCBLR, PICK). LISP 1.5 eliminates this, leaving it to the user to provide the necessary functions.

    d) GO was a SUBR in LISP 1 and evaluated the argument in the usual way.

    e) The introduction of dotted pairs restructures the A-list, and with it, the function SASSOC.

    f) The function TRACKLIST is now named TRACE.

    g) CSET and CSETQ are introduced in LISP 1.5.

h) The function ERRORSET is introduced for Abraham's proof verifier in LISP 1.5.

4. Changes in the Compiler System:

   a) LAP has a new structure: markers are atoms on the program level, and commands have one element less.

   b) The LISP 1.5 compiler is essentially new. It generates functions that allocate temporary variable space in the cellar.

5. Changes in the Processing System Overall:

   a) In LISP 1, a kind of APPLYQUOTE was the main function: the user had to provide triplets—function, argument, and A-list. In LISP 1.5, EVALQUOTE is the main function, and the user must provide doublets of expressions: function and argument list.

   b) In LISP 1, it was stated that atoms have an association list, and the system works with a property list. In LISP 1.5, the notations have been swapped.

   c) The Flexowriter system has been eliminated in LISP 1.5.

   d) The object list is organized as a hash list.

   e) The error messages have been largely changed.

   f) Each PROG in LISP 1 had to be terminated with a RETURN.

   g) A joker made sure that at the beginning and end of a run of the LISP 1.5 system, a greeting and end time indication in the form of a Lewis Carroll quote occurred.

It should not go unnoticed that from about 90 functions, there were now around 140. Naturally, a multitude of details in the implementation had also changed.

The LISP 1.5 Manual [MCC62c] is considered today as the classic description of the LISP language. It has established itself over its 1960 predecessor due to changes, improvements, and the absence of a successor. The short period of use of the LISP 1 system on the IBM 704 has been almost completely overshadowed by this system. Today, there are voices equating LISP 1 with the so-called "pure LISP" (LISP without Prog feature and side effects). That this is not correct is evident with a brief look at the parts of the system indicated by McCarthy in [MCC60c], which includes the Prog feature. As McCarthy reported [MCC76], there were sequential elements and assignments from the beginning.

The term "pure LISP" is used only in the LISP 1.5 Manual ([MCC62c], p.14) ("the pure theory of LISP"). Here, it refers to the complex consisting of M-language, S-expressions, the translation process from M to S-expressions, and the theoretical universal evaluation program APPLY/EVAL. However, "...the reader is warned, as these functions are only pedagogical tools...".

Another terminological problem is the designation LISP 1.5. According to McCarthy, this name was introduced only in 1962 when the variant for the IBM 7090 was already running. "The name came with the new manual" [MCC76]. Apparently, it was felt that the new system was significantly superior to the old one. However, the name LISP 2 was not chosen because the ideas for LISP 2 had already been developed [MCC76]: "We already knew that we wanted to write LISP 2." The ideas for LISP 2 seem to have gone in a specific direction from the beginning due to the emergence of ALGOL. However, the name LISP 2 is not found in early publications (see, however, [ED61]), and it is only traceable from 1963 onwards (e.g., [MCC63f.]).

The LISP system on the IBM 7090 provided users with extensive capabilities. It was equipped with a rudimentary batch processing system (OVERLORD), allowing the processing of closed packages of function calls and monitoring the saving or restoration of the system on magnetic tape.

The system was significantly shaped by the input format (doublets of function names and arguments) and by the interpreter based on A-list processing. The work with properties and P-lists was only partially implemented, as evidenced by the EXPR form (i.e., not as machine programs) of the most important P-list functions ('DEFINE, DEFLIST, CSET, CSETQ, FLAG,' etc.). ATTRIB added the latest property to the end of the P-list.

The total number of implemented functions was already so enormous, with 113 elements, that the entire program system comprised 170,000 (octal) words. The revised compiler is the origin of a whole family of compilers, some of which are still in use today.

Until the seventies, every LISP implementation drew essential parts of its design philosophy from the LISP 1.5 system on the IBM 7090 at MIT.

This brief characterization is sufficient at this point. Chapter 5 will describe the LISP 1.5 implementation in more detail.

The LISP 1.5 system exerted a strong attraction on all scientists working in the Cambridge area on language implementation and artificial intelligence problems. Programs written in LISP opened up an entirely new area of programming, and the creators of these programs did not want to miss the highly advantageous LISP environment. If they had to change their workplace, they took the system with them. The circle around McCarthy grew rapidly in a short time. Many of today's leading scientists in the field of artificial intelligence contributed to the LISP 1.5 implementation at that time. The actual main work, of course, had been done by the two programmers.

According to the information in the manual [MCC62c] [p. iv], the following areas of work are known: Interpreter—S.B. Russell and D.J. Edwards; Input and output—J. McCarthy, K. Maling, D.J. Edwards, and P.W. Abrahams; Garbage Collection and Arithmetic—D.J. Edwards; Compiler—T.P. Hart and M.I. Levin. In the stimulating environment, mention should be made of M.L. Minsky, B. Raphael, L. Hodes, D.M.R. Park, D.C. Luckham, D.G. Bobrow, J.R. Slagle, N. Rochester, C.E. Shannon, R.K. Brayton, L. Kleinrock, S.H. Goldberg, P. Fox, H. Rogers Jr., H. Teager, D.A. Dawson, E.L. Ivie, P.G. Jessen, and U. Schimony.

When McCarthy left MIT in November 1962 to follow a call to Stanford, LISP began to gain recognition. Implementations on other computers were made, and largely due to participants in sharing systems and the interactive operation made possible by them, the features of LISP continued to prove themselves.

The beginning had been made.

3 Local distribution of LISP 1962–1966

Initially, McCarthy (with his followers) worked in the "Computer Science Division" of the Department of Mathematics, and later (1964) the department split off as an independent department. The initial period was dedicated to both pursuing theoretical work [MCC63c,d] and adopting the LISP system, focusing on its improvement. When the Stanford Artificial Intelligence Project opened in June 1963, McCarthy had six staff members [EAR73].

The next stage of LISP development is significantly marked by work in the field of time-sharing systems.

The idea of making a computer available to multiple users by dividing the central processing unit's time into equal portions emerged around 1958-1959 [STR59]. McCarthy had already started implementing related ideas in 1959. Even on the IBM704, a system had been built based on inserting small program pieces between the jobs of a batch processing system. The operating system inquired whether input was required from a terminal (usually a simple teletype or Flexowriter) upon job completion. If the answer was positive, it served an intermediate program. If jobs in the batch were limited to a processing time of 1 minute, the working programmer could expect an acceptable response time. However, the program for interactive work had to be constantly in main memory. Fortunately, the conditions were met: customer programs only expected an available space of 8192 words, while the memory had just been increased to 32768 words [MCC78a]. After Edwards and Luckham [MCC60a] wrote the necessary routines in early 1960, McCarthy could study the benefits of online work. Apparently, the initially modest success was so encouraging that McCarthy strongly advocated for time-sharing. The LISP1 system served to demonstrate the advantages of interactive work, even though it didn't actually use time-sharing but rather "stole" otherwise lost time. Therefore, this rudimentary procedure was also called Time-Stealing [MCC78a].

Due to McCarthy's initiative, interest in time-sharing grew at MIT, leading to the formation of study groups addressing this issue. McCarthy led the MIT Computer Working Committee and was a member of a Preliminary Study Committee. He was even offered the directorship of the MIT Computer Center to directly oversee the implementation of a time-sharing system, but he remained committed to Artificial Intelligence. Thus, H. Teager, the leader of the second study group, became the organizer of MIT's work on time-sharing [GL67].

However, McCarthy's ideas seem somewhat unrealistic; in any case, F.J. Corbato preferred a modest design and independently started implementing

his experimental time-sharing system in 1960. The first experimental version [COR62a] of this system ran on an IBM7090 in 1961. Experience was quickly gained, and the LISP system also ran, without significant changes, under the control of this operating system. By the end of 1962, the well-known Compatible Time-Sharing System (CTSS) had evolved from an experiment [COR62b], ultimately running on the IBM7094.

McCarthy not only gave popular lectures on time-sharing systems for the pdp-1 at BBN [MCC63b].

The programming work was done under the guidance of E. Fredkin, an employee of the company, Boilen. Since the computer was very small, accommodating only 8 users, only one program could be run at a time. Fredkin considered the outsourcing of the entire program to the drum and the loading of the new program as a "poor substitute solution." Fortunately, both could be done with a single operation. The technique of swapping, thus discovered, became the standard solution in almost all time-sharing systems. Although it had been decided upon McCarthy's departure from MIT that the further development and maintenance of the LISP system would now take place in Stanford [MIN63a], the adjustments required by the time-sharing operation necessitated a lot of work at MIT. The daily practical online work with LISP in CTSS demonstrated the excellent suitability of the language and programming system for interactive work, as programs, input, and output data were all in the same language.

The discovery of errors and the pursuit of desires arising from the use of the system led to constant maintenance and revision at MIT. Improved versions were frequently put into service [ED64a, MART64]. Among the extensions, the introduction of macros triggered by Hart [HART63b] must be mentioned. Remarkable in the implementation of the macro concept was that LISP macros were treated as special functions during interpretation, with the expansion of a macro being performed by evaluating the expression to be evaluated. Macro expansion during compilation did not differ from the usual procedure in other programming languages.

A second significant work on LISP was the effort to exclude secondary storage media from the work. Edwards [ED63] provides a good overview of the state of affairs in "Secondary Storage in LISP." The author reports on magnetic tape work in the original 7090-LISP and the extended use of these devices through LISP libraries, where functions could exist both in source format (S-expressions) and in LAP format (compiled functions).

For the arrangement of compiled functions in main memory, Edwards suggests the ring buffer, which allows each loaded function to remain in memory for a certain time. Since active functions should be protected before removal, Edwards expected this solution to allow functions to be jumped to and executed at nearly full machine speed in typical LISP situations.

By combining this idea with Minsky's concept of a compacting garbage collection using secondary storage media [MIN63c], the size ratios of the ring buffer and list storage could be changed according to dynamic requirements.

A third possibility Edwards saw to deal with extensive sets of functions was the declaration of functions to be used "soon," which would then be loaded as

needed.

For the history of LISP, Edwards' explanations of paging techniques are of highest interest. According to his information, a pdp-1 at MIT had been equipped with virtual memory in 1963 (McCarthy and Fredkin attribute the project to J. Dennis [FRE77]). The pdp-1, which had 4096 words of 18 bits each, normally provided only about 800 words for list storage. By swapping pages with a drum that could hold 88K words, this memory was significantly expanded. Edwards criticizes the still imperfect techniques of software paging for cross-page references and the resulting significant slowing down of the computation process. Unfortunately, he does not mention the developers of the system and provides no other data.

The often-used methods today of outsourcing intermediate results through symbolic output to work files, from which they can be reread if needed, are viewed skeptically by Edwards. The reasons for this lie in the too high requirements the author places on this variant of using secondary storage. From today's perspective, limiting this method to normal list structures seems entirely sufficient. The cyclic and BLAM (Berkeley Lisp Abstract Machine) lists ([ED63], p. 5), with which Edwards deals, occur relatively rarely.

As a last option for working with external storage media, Edwards describes the automatic compacting list structure main memory dump, attempting to temporarily store at least part of the active list memory on secondary storage.

Overall, this work provides a good overview of the LISP know-how of that time. It was intended for the 1st International LISP Congress, which took place from December 30, 1963, to January 3, 1964, in Mexico City. This conference was intended to be the first international forum for the discussion of the language and its implementation. However, only the two works by Minsky and Edwards are known from the materials of this conference.

That such a meeting was meaningful is demonstrated by the increased activity regarding LISP around the turn of 1963 to 1964. In Cambridge, work on LISP implementation was carried out in several places; at Stanford, the old system was revised, and other centers where LISP was being worked on were emerging. An article by Borrow and Raphael on languages for symbol manipulation provides a reasonably good but incomplete overview of the situation at the end of 1963 [BO64c].

```
CENTRO DE CALCULO ELECTRONICO.

CIUDAD DE MEXICO

Mexico, 20, D. F.
FIRST INTERNATIONAL LISP CONFERENCE

December 30, 1963 - January 4, 1964
Preliminary List of Participants and papers

CENTRO DE CALCULO ELECTRONICO.
```

```
CIUDAD DE MEXICO

 Mexico, 20, D. F.
FIRST INTERNATIONAL LISP CONFERENCE

 December 30, 1963 - January 4, 1964
Preliminary List of Participants and papers

\begin{center}
DULOCK, VICTOR.- LISP. Applications to Symmetric group, Dirac groups and Lie algebras.
EDWARDS, DANIEL.- Secondary Storage in LISP.
EVANS, THOMAS.- Character String manipulation in LISP.
HART, TIMOTHY.- Macro Instructions for LISP.
HAWKINSON, LOWELL.- Data structures and arrangements in LISP.
HEARN, ANTHONY C.- LISP. Computation of Feynman Graphs.
LEVIN, MICHEL.- Algebraic Compiler with LISP.
Mc. CARTHY, JOHN.- The LISP. 2 Compiler
Mc. INTOSH, HAROLD V.- The use of operator predicates in LISP.
MINSKY, MARVIN.- Garbage Collector methods
THOMAS, BILLY S.- Use of arrays in LISP. Group theory Programms.
WEIZENBAUM, JOSEPH.- Open Ended compilation.
WILLIAMS, JOSEPH.- A LISP. Page plotter.
WOOLDRIDGE, DEAN.- An Algebraic Simplify Program in LISP.
YATES, ROBERT.- LISP. Group Analysis programs in LISP. Compiler for a variable word machine
RUSSELL Debugging aids
VERHOVSKY Fns analoguous and similar December 15. 1963
\end{center}
```

Fig. 8. Program of the 1st International LISP Conference

This work contains no mention of H. McIntosh's efforts to disseminate LISP. Around 1960, he encountered this programming language at MIT and initially implemented a version of LISP at the University of Florida, where he conducted experiments. Later, in early 1963, he became a professor at the University of Mexico in Mexico City. There, he gathered a group of students to tackle problems in symbolic information processing. By the end of 1963, they had an IBM709 and used either the original LISP 1.5 version from Cambridge or their own variant. McIntosh's collaborators at that time included A. Guzman and L. Hawkinson.

The latter was already not a novice in LISP. He had encountered LISP around 1962 and had programmed a system of interpreter and compiler for an IBM709 at Yale University. He followed McIntosh to the University of Florida in Mexico, where he worked on another implementation. Thus, the conference location provided a stimulating environment for the further development of LISP.

There is good information available about four of the systems built in Cambridge at that time. All four had some influence on the advancement of implementation techniques and the external form of the LISP language.

The LISP system developed by Hart and T.G. Evans for an M460 [TH58] at the Air Force Cambridge Research Laboratories [HART64a] stood out due to its innovative development technology at that time. Initially, the main specifications for internal structures were made, and the garbage collector was programmed. Then, all necessary machine routines were written (`CAR, CDR, CONS, RPLACA, ATOM, GENSYM, NUMBERP, PLUS, TIMES, GREATERP, EQP, READ, PRIN1, TERPRI, PUN1, TERPUN, EVALQUOTE, SET, VALUE`, and some others). The main effort involved creating a LISP compiler in LISP (by modifying the 7090 LISP compiler) and using it to translate the LISP interpreter and the remaining basic functions for the new machine. This work was done on the IBM7090. The result was a large machine program added to the previously written individual routines. Finally, the compiler used for transport was also adopted into the new environment.

However, the M460-LISP was not only a technological novelty. It had three essential differences from LISP 1.5 on the IBM7090, based on forward-thinking ideas: The interpretation was based on the A-list shallow access scheme (see p. 208) and worked only with stack memory and value cells.

To simplify the memory structure and perform simple string processing, names and numbers were implemented as lists of numbers or number pieces (character string lists). Apart from the compiled programs, the memory did not need to be divided into separate areas.

The character lists required a new syntax element: to read S-expressions that contained such character literals, it was agreed to announce each character literal with a slash, for example,

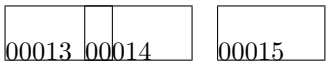

Fig. 9. The character string list `(/a/b/c)` in M460-LISP ([HART64a], p. 197)

Another difference, though not visible to the system user, was the compacting garbage collector. As far as known, this scheme was used here for the first time.

In the first phase, active cells were marked. Since the pointers occupied the entire word, bit tables had to be used to indicate the use of a cell. The starting areas for marking were compiled programs and stack memory (object list is not mentioned). Active cells were counted during marking.

Then, in the 2nd phase, the memory was divided into the area to be moved and the fixed area; in the fixed area, free cells were identified.

In the 3rd phase, the cells of the area to be moved were moved into these free cells. A pointer to the new location was placed in the old cell.

Finally, in the last phase, the entire memory had to be searched for references to moved cells in the area to be moved, and the new address had to be substituted.

Shortly after the completion of M460 LISP, Saunders began building a LISP system for the Q-32 computer[9] for the System Development Corporation, Santa Monica. This system heavily relied on the experiences and results obtained while working with the M460. Saunders mentions specific assistance from Hart, Edwards, Levin of Cambridge, Russell, and McCarthy from Stanford.

The Q-32-LISP [SAU64b] essentially adhered to the LISP 1.5 specifications. The macro system developed by Hart was part of the interpreter. A new feature of Q-32-LISP was that all functions defined with DEFINE and all macros defined with MACRO were immediately compiled. The compiler, therefore, no longer used the common variables necessary in the 7090-LISP 1.5 to coordinate the work of the interpreter and compiled programs but only used SPECIAL variables. These also replaced the APVAL sizes.

The garbage collection, as in M460-LISP, was based on a compacting scheme.

The technology for creating Q-32-LISP was very similar to that used in M460-LISP. According to Saunders [Sau64b], it was programmed half in the SCAMP assembly language and half in LISP. The LISP part, which consisted of about 1900 cards, was compiled on the IBM7090—a process that took 40 minutes.

The part written in assembly language included the basic functions, a program part that read and processed the result of compilation produced on the IBM7090 (on magnetic tape), and finally, a part that could execute main memory dumps in the Q-32 system.

The compiler, (LAP)-assembler, and a multitude of LISP functions were included in the LISP part. This part of the Q-32-LISP programs is fully described in [SAU64c].

In general, Q-32-LISP (see also [KAM64, WEIS65]) was considered well-described [AB68], and the published compiler likely served as a model for many later-developed LISP systems.

Q-32-LISP continued to evolve (Saunders reported on two versions, Mark I and II) and was used until the end of 1967. It had significant importance as the base system for CONVERT [GUZ66] and LISP2 [AB66b], as the Q-32 could be accessed remotely, even from Mexico City.

For a similar reason to Q-32-LISP, pdp-1-Basic-LISP [DEU64] became known: its description not only provided general properties but also published the entire

[9] exact designation AN/FSQ-32/V, according to McCarthy[MCC76] a military computer; only one ... was used for scientific work. The machine had a 64K memory of 48-bit words and was essentially a better machine than the 7090. However, it never received more memory and was difficult to maintain. According to Glauthier [GL67], the Q-32 was an IBM machine derived from the 7090.

Important for LISP was that this computer operated in time-sharing mode and was also hardware-adapted to related tasks. However, the software paging is said to have been quite slow.

interpreter as an assembly program.

This system was created by L.P. Deutsch, who studied at MIT in 1963 but focused more on private programming interests than on formal studies. The first version was completed in October 1963, and a further improvement was finished in March 1964. Perhaps this system also served as the basis for pdp-1-LISP by Edwards [ED63] in virtual memory.

pdp-1-Basic-LISP used a minimum of 2000 words and could occupy the entire memory in an extended 16k pdp-1. Of course, not much could be done in such a limited environment. The input media were teletypes and punch card readers; output was usually on the teletype. During input, the user had to pay closer attention to the format (specific spacing requirements) than in LISP 1.5, as the input program was not as flexible.

Deutsch's system was the first to consistently move away from the higher-level EVALQUOTE function and replaced it with EVAL. This led to the somewhat inconvenient necessity of quoting arguments at the top level. Another invention by Deutsch is the slash character. In pdp-1-LISP, after the "–" (overbar), any character could appear and would be treated as a letter.

This LISP system was not the last on a pdp-1, as this machine was characterized by great flexibility and an unusually wide range of peripherals for that time.

The relationships that the Digital Equipment Corporation (DEC) had established with MIT (due to proximity) were always very close. In several cases, new computers were delivered to MIT very early. An important example of this close collaboration was the development of a LISP system for the pdp-6.

In the spring of 1964, a group (including Greenblatt and Nelson) of DEC employees and members of the Technical Railroad Club, a student interest group at MIT[10], began designing, programming, and implementing an interpreter and a minimal LISP system for the pdp-6 [WHI70]. This unlikely step (there was no operational pdp-6 at that time!) was made possible by using a well-equipped pdp-4 that handled program management and modification, assembly, and later output. Whether the program also ran or was simulated on the pdp-4 is not known. It should be noted that both machines are not very similar, as they belong to two separate branches of the DEC computer family (pdp-1, -4, -7, etc.; pdp-5, -8, -12, etc.; pdp-6, -10).

When the first pdp-6 computer [BEL78] was delivered to MIT in mid-1964, the new system was the first program to run on the facility, and it served simultaneously as a means to test the machine. According to J.L. White [WHI70] [S.v], it was the first program ever to run on a pdp-6.

Not much is known about the original version. The design ideas were supposed to have been taken from the CTSS version on the IBM7094 and pdp-1-Basic-LISP. However, for a long time, pdp-6-LISP must have been fully compatible with LISP 1.5. Initially, it was more extended than changed; in particular, input and output capabilities, error messages (which until then consisted of old

[10]Members included Russell, Saunders, and A. Kotok, who is said to have written the first chess program in LISP

abbreviations of LISP 1.5 from 1962—see [MCC62c][p.32]), and various subroutines improved significantly. Many old errors were removed. Some errors had to be indicated as known but not yet eliminated [MOS66][p.7].

The new functions were mainly of a numerical nature (e^x, ln x, NUMVAL, EXPT), enabling better work with parts of the LISP system (e.g., the user could determine the number of free cells in the area of compiled functions, set or query the timer, etc.) or allowing new actions with lists. Notable here is the EXPLODE function, which, starting from any S-expression, provides a list of characters as they appear when printing the expression on paper (credited to W. Martin).

This new CTSS-LISP seems to have been used only at MIT. It also appears to be the last link in the family of LISP systems built at MIT for IBM 704-IBM 7094 computers. With the advent of the pdp-6, interest turned to this machine, partly designed with a focus on applications for list processing.

The results of the work at Stanford University were widely disseminated. Among the areas that McCarthy had initiated, in addition to formula manipulation [EN63b, RUS63b], the development of the pdp-1 time-sharing system was the further development of the LISP1.5 system, which had been transferred to Stanford.

In addition to small additions (time functions [EN63a]) and minor changes (such as to the compiler [WOL64a]), considerations were made about enhancing error-checking aids in the LISP system [RUS63a]. Entirely new were some functions intended for working with files on disk drives. They allowed the temporary output of S-expressions. However, users had to manage their files to find the right one to read in the sequentially built sequences of S-expressions [WOL64b].

This system, temporarily called LISP1.55, was handed over to the IBM user community SHARE. With some additional small additions, it is described in the 1965 edition of the manual [MCC62c] as Appendix I. Its completion is dated February 1965.

McCarthy and his colleagues had an influence on the implementation of Q-32-LISP. Occasionally, this system may have been used from Stanford as well. Together with MIT, efforts were made on the design of LISP2, involving frequent conferences and the creation of position papers [MIT64]. A significant development for the history of LISP began around 1963 with A.C. Hearn's work on his LISP program for calculating cross-sections of elementary particle reactions using Feynman diagrams. Hearn, working as a physicist at that time, developed a keen interest in symbolic manipulation and started building one of the most important (and probably most used) formula manipulation systems in 1965.

However, for some of the tasks and most projects, the LISP system on the IBM 7090 was not powerful enough, and the facility itself was too small. The situation was set to change fundamentally in 1966 with the introduction of a pdp-6 (cf. [MCC65]). Between 1963 and 1964, students at Stanford created a small LISP system for the B5000. It had to be programmed in ALGOL, the machine language of this computer.

With Bobrow's departure from MIT and the start of his work at the data

processing company BBN (Bolt, Beranek & Newman), the third LISP center in the USA was established. Bobrow's main focus at MIT had been on symbolic manipulation [BO63a] and natural language processing [BO64a]. However, in addition to these ongoing areas of interest, he had always been involved in developing tools necessary for these fields (see, e.g., [BO64c, d]).

He had early on developed the opinion that symbolic manipulation must be supported by tools of pattern matching but missed these tools in LISP (although he found them in COMIT [YN61, 62]). Therefore, he advocated equipping the "new" LISP version (i.e., LISP2) with these tools [BO64b, 65] and had a significant role in the design and possibly also in the implementation of LISP2 (at least in the early days).

BBN had a department in 1964 that extensively dealt with LISP. In early that year, a large experiment to compare programming languages was initiated [BE64]: Twelve selected programmers, considered experts in the languages they used, were tasked with programming seven different problems. Parameters over time for program design, testing, etc., were recorded. Surprisingly, LISP performed very well.

ALGOL experts required 2 hours, the FORTRAN expert 1 hour, the COBOL expert 2 hours, and the LISP expert (Hart from MIT) 1 hour for programming. To test the program, the ALGOL programmer needed 4 hours, the FORTRAN programmer 15 hours (in CTSS interactive mode!), the COBOL programmer 6 hours, and Hart with the MIT LISP system only 1 hour.

The operation of the LISP system under CTSS ensured that Hart was significantly ahead in the overall processing time: While ALGOL took 7 days, FORTRAN took 16 hours (CTSS interactive!), and COBOL took 5 weeks for processing, Hart with the MIT LISP system delivered complete solutions just 4 hours after receiving the tasks.

This proved that LISP was not only suitable for designing and testing programs but also easily accessible to programmers.

One of the first projects that Bobrow worked on at BBN was the development of a new LISP system for a pdp-1 installed there. As mentioned earlier, this computer was small (16K words of 18 bits) and reasonably fast (access time 5μs), but it was one of the first real-time computers. The pdp-1 at BBN was equipped with excellent peripherals (card reader and punch, teletype with reader and punch, magnetic tape devices, light pen and a graphic tablet; also an 88K drum with an access time of 17 ms).

Together with D.L. Murphy, Bobrow designed a LISP system based on a virtual memory concept [BO66a, BO66b, BO68a]. This system became well-known, especially for its memory management techniques. It likely influenced the operating system TENEX [BO72].

With extremely limited memory, it was essential to ensure that only the most critical programs were permanently in memory: interrupt handling, time-sharing supervisor, and the paging system (4K). An additional 4K-word area served as an overlay area for interpreter and runtime routines of compiled functions, for non-"critical" LISP subprograms for list processing, input and output programs, garbage collection, and the initial program.

To efficiently process compiled functions, a ring buffer was created and managed using a *transfer vector*. Due to the limited size of the ring buffer (3400 words), a program calling subfunctions could be pushed out of the ring buffer by these subfunctions. To prevent such an undesirable situation, function communities were established, i.e., sets of functions calling each other ([BO66a], p.6).

The basic idea of the virtual memory was the allocation of address ranges and data types. It should always be possible to derive the type of an object from the pointer pointing to that object. This was achieved by dividing atom heads into `PRINT-NAMES, PRINT-NAME-POINTERS, FUNCTION-CELLS, PROPERTY-LIST-CELLS`, and VALUE-CELLS. The pointer to an atom was the pointer to the value cell. From this, the other parts could be derived.

Only large numbers were represented in a numeric storage area. All numbers between -32767 and +32767 were directly represented by pointers, allowing comparisons without drum accesses and saving space ([BO66a], p.11).

When constructing list structures, CONS followed heuristic rules to determine the page onto which a new cell should come ([BO66a], p.12). Garbage collection consisted of a local manager for pages and an apparently seldom-used dump program for compacted structures [BO66c].

With these specifications, the pdp1-LISP system was undoubtedly an interesting system. Runtime measurements showed strong dependencies on data structures, causing a program to wait for the drum between 10% and 50% of its processing time ([BO66a], p.16ff). Apparently, the experiences with heuristic rules for limiting the spread of list structures across pages were so good that they were retained later. It is still an open problem whether a consistently compacting garbage collection or a CONS working according to Bobrow's ideas is the better solution. Measurements and concrete comparisons are not available.

Although the BBN pdp-1-LISP system had nothing to do with Deutsch's system (cf. p.199), Deutsch was involved in the work. He advised Bobrow in designing the drum memory and wrote one of the first compiler versions [BO76]. Shortly after that, Deutsch was tasked with building another LISP system on a computer that Bobrow would later also use as his workstation.

From 1964 to 1965, the SDS 930 at the University of California in Berkeley was developed into the SDS 940 paging machine [PIRT66], which was supposed to be adapted for time-sharing. Deutsch, who was working there at the time, implemented a LISP system [DEU66][11]. He was also involved in text processing [MOE65] in dialogue. Based on the experiences gathered in this context, he was a crucial support in transferring the BBN LISP system to the SDS 940 and in designing the software paging system [BO76, DEU76].

The aforementioned comparison result made LISP attractive among programmers dealing with non-numeric problems. Thanks to the popularizing article by E.C. Berkeley [BE64] and, above all, the anthology "The Programming Language LISP: Its Operation and Applications," jointly edited by Bobrow and

[11] Deutsch was associated with all the work in Berkeley. He participated in the development of the time-sharing system and many other programs. For the non-upgraded SDS 930, he also built a LISP system, which certainly significantly influenced the system on the 940 [DEU65]

Berkeley [BB64], LISP became not only known in America but also drew attention abroad. Although LISP implementations had started in several places outside the USA in the course of 1966, the IFIP Conference on Languages for Symbol Manipulation in Pisa from September 5 to 9, 1966, marked the real international breakthrough [BO68b].

W.L. van der Poel's first LISP system was said to be functional at that time [POE78], J. Kent's thesis was almost finished, and the LISP system modeled after LISP1.5 on the CDC3600 was operational. Implementation of LISP also began in Poland after this conference.

In Pisa, not only were specific implementation techniques discussed, but program systems for formula manipulation, such as MATHLAB, were also presented. C. Engelman's [ENG66] remarks about the convenience of working with LISP spoke for themselves and were excellent advertisements. Thus, it didn't take long before LISP became a widely used programming language.

4 Worldwide distribution of LISP 1966–1978

I

In the years after 1966, the LISP scene has undergone a transformation. Whereas before, development was characterized by the spread of LISP to ever-new computers while retaining essential characteristics and the growing utilization or appropriation of the language's characteristics, along with reconciliation with the language form, one can now distinguish two quite different forms of further development:

On the one hand, we still see, to a large extent, the previous spread continuing: new implementations for already-used or other computers are still being established, and new user groups are initiating LISP implementation in their work environment. Here, the geographical dimensions have now expanded – Europe and Japan are being encompassed by LISP.

On the other hand, the more advanced form of LISP development can now be clearly divided into lines. Although possibly each LISP system is different from others, there are clearly definable groups led by a guiding system. The development of the guiding system is driven over the years by a consistently stable core of system programmers and maintainers. Where these individuals lose interest or change jobs, the development of the guiding system stops or, in rare cases, is taken over by others.

It is quite difficult to select from the wealth of material what is not only reducible to local program improvements or extensions by expanding the functional basis but also shapes the outward appearance of the language. The transition here in LISP is certainly fluid (as it tends to be in programming) – one can rightly understand LISP as a programming system, a mere collection of basic functions.

However, natural language cannot be dismissed simply as a collection of vocabulary either. Whether the programming language is embodied by syntactic sugar or significantly shaped by it also appears quite questionable, to express the

other extreme. Whether, therefore, "true" language development needs to be separated necessarily from vocabulary extensions and must offer new syntactic means for programming languages is again quite problematic.

Nevertheless, we will attempt to separate those functional innovations that are only a completion of certain basic sets (such as the inclusion of trigonometric functions) from those that express a certain change in language understanding and usage. Only the latter will deserve attention.

Likewise, many improvements in implementation techniques may not be mentioned. The painstaking work associated with them, however, will not be sufficiently appreciated.

To systematize the development of the language advanced by the guiding systems across these lines, the following important driving forces shall be named:

1. The daily use of the language led to needs for test aids, automatic error correctors, support systems for working with large data sets of LISP programs, and other tools for interactive dialog work.

2. The simultaneously evolving scientific discipline of Artificial Intelligence (McCarthy: "The science should be called Cognology, the goal is to develop programs that show artificial intelligence" [MCC77]) led to new problems, for the solution of which researchers demanded new formulation and coping tools [BO73b]. Although LISP not only remained unchanged at its core, some extensions have been able to satisfy these demands. Others have remained open or have been satisfied within the so-called higher programming languages that build on LISP.

3. The use of LISP for the implementation of large formula manipulation systems, partially driven and continued by the most active LISP maintainers, led to contradictions between the suitability of LISP for symbol manipulation and that for numerical computation. In addition, these researchers had to provide tools to their customers, i.e., the actual users, physicists, and engineers who did not know LISP, which in some ways represent a continuation of the user aids from a).

4. A "big theme" in the further development of LISP is the use of this language for the implementation of other programming languages. Although the focus has somewhat shifted from extensible languages [BO64d], the tendency still exists. LISP has benefited from these projects to some extent (besides the obvious benefit of being used) since various concepts from higher languages have drifted into the base.

We will address metasystems in Section 4.8.

To organize the material on the history of LISP during these 12 years, the development of LISP will be treated separately as much as possible, and the development influenced by the guiding system. Otherwise, a geographical approach will be taken. Thus, we will treat Europe, Japan, and other countries separately from the USA.

4.1 MacLISP

Triggered by the author's inquiries, the long-time LISP expert J.L. White published a work on the development of MacLISP, making essential historical material available [WHI77].

In early 1966, MIT had two LISP systems: the continuously improved LISP1.5 system on the CTSS running on an IBM7094, and the new LISP1.5 system on the pdp-6. During the period between 1965 and 1966, there was a change in staff. Almost all previous researchers had left MIT or shifted to other areas, except for Minsky. In their place, individuals such as Moses, Martin, Greenblatt, Nelson, P. Samson, Teitelman, and others worked on LISP implementation. A distinct group of researchers in the Artificial Intelligence Laboratory began to focus on formula manipulation.

Inspired by experiences with large LISP program systems and facilitated by the development of a new compiler (for the pdp-6), discussions arose about abandoning variable binding using the association list. Instead, a stack storage model similar to block-oriented programming languages was considered. According to White [WHI77][p.2], it was Greenblatt who, in emulation of FORTRAN, suggested inserting current variable values directly into the atom representation, eliminating the need for any value search. Old variable values were centrally saved in a special stack storage called the Special Push-Down Stack.

This new approach resulted in significant speed advantages. However, the FUNARG model for handling global variables could no longer be fully implemented. Nevertheless, this seemed initially to be a non-significant loss.

Presumably, the work on compiled functions was the starting point. In the complex mix of APVAL, COMMON, and SPECIAL variables, much work was executed in parallel. Someone overseeing the entire system's work could easily determine that only one such type was actually necessary. Initially, the most important type was used: all interpreted and free compiled variables were automatically treated as SPECIAL variables, ensuring complete communication between compiled and interpreted functions.

The implementation of this approach initially had the disadvantage that too many SPECIAL variables occurred when interpreted first. The representation of the value cell (SPECIAL cell) as an element of the P-list also had drawbacks, although it was partially enforced by the pdp-6 word structure. When the P-list was modified, a small search operation in this list might be necessary. It became customary to refer to a LISP system operating with this type of variable binding (shallow access according to Moses [MOS70]) as LISP1.6.

The first LISP1.6 system, functional around June 1966 [SAMS66], included many other changes. For the user, the convenience provided by the new system, the quantity of functions, and the services it offered were likely more important than the question of variable binding.

After apparent progress through further stages of improvement and substantial revision [PDP6a], a certain stability seemed to be achieved around early 1967 [PDP6b, PDP6c]. This version essentially persisted until around 1970 [WHI70].

Typical new features in the pdp-6 LISP included associated functions with

any number of arguments, and with two arguments, APPEND and *APPEND, GREATERP and *GREAT, LESSP and *LESS, etc. Additionally, character manipulation functions EXPLODE and EXPLODEC paralleled the print functions PRINT1 and PRINC. Input and output functions UREAD, UWRITE, UFILE, and UKILL, the multipurpose function BOOLE (including LOGAND, LOGOR, LOGXOR, etc.), and functions for P-list operations DEFPROP, REMPROP, PUTPROP, GET, and GETL were introduced.

Adopted from pdp-1 LISP was the Slashifier, denoted as a character (initially "/"), ensuring that the next character was considered a letter character and thus a valid part of a LISP atom. This made the old $-bracket of LISP1.5 and all kinds of symbolic output streams of the system readable again.

The year 1967 was marked by the development of the pdp-6 operating system by the AI Laboratory [EAS72] and the transition from CTSS to MULTICS at the MIT Data Center [COR65]. The developers of the LISP1.6 system, to which White had joined by then, initially faced the task of clarifying and optimizing relations with the operating system. This resulted in the ability for the LISP system to run both within and outside the Time-Sharing System, with connections to other programming languages, especially assembler programs [SILV 67]. The most extensive work, however, focused on new input and output capabilities, introducing both new devices (television cameras, plotters, displays) and improved access methods.

By early 1968, with the delivery of the pdp-10 and the launch of MULTICS on the GE634, device technology reached a level that would remain stable for years. At the AI Laboratory, the development of the incompatible Time-Sharing System ITS was completed in response to the demands of working with robots and vision systems.

For the GE634, an initial LISP system [YO67] was developed in 1967 by J.C. Yochelson as part of his BS degree. The experiences gained with this system, particularly regarding memory management [FEN69], were crucial prerequisites for the later system that became fully compatible with the pdp-10 system.

With the environment for LISP systems now somewhat stable, a new stage of language development began at MIT. The driving forces were the scientists working on the formulation of the formula manipulation system MACSYMA (Moses, Martin, R.J. Fateman, J.P. Golden, and a number of students), as well as those considering new programming languages for expressing problems in Artificial Intelligence (C.E. Hewitt, G.J. Sussman, T.L. Winograd, I.P. Goldstein, V. Pratt, Greenblatt, and T.L. Binford, along with some students).

The first significant new idea, related to the currently widely discussed "extensible" language concepts, was the introduction of macro characters. Each character could be associated with a program that typically reads additional character strings and produces an expression used for the macro character and the concluding character string. The corresponding program is, of course, written in LISP. This construction, proposed and implemented by White, made LISP a dynamically syntactically alterable language. Early use cases involved the ' as a quote character, the / as a Slashifier, and later the use of prefix characters in Sussman's MICROPLANNER implementation [SUS70]. Around the

same time, White also introduced the variable base for integers, allowing these numbers to be based on bases between 2 and 36.

In 1968, plans were initiated for two significant developments, gradually realized: At least until 1969, and still nearly everywhere outside MIT, it was believed that LISP was only suitable for non-numeric problems. Numeric calculations were considered possible, but with disproportionately long processing times. For the development of the MACSYMA system, which aimed to allow both symbolic simplifications and numerical evaluations of formulas, the speed of execution of programs written in FORTRAN posed a serious problem (White [WHI77] speaks of a 1:100 ratio).

Efforts were made to provide better arithmetic capabilities. The internal representation of numbers was reduced by two-thirds from the LISP1.6 conception (numbers having a similar structure to literal atoms). White eliminated the temporarily introduced small integers (referred to as INUM in LISP1.6) for uniformity.

A range of new arithmetic fundamental functions was introduced, including both new versions of existing functions, working faster due to assumed type purity (only integers or only floating-point numbers), and completely new functions (square roots, greatest common divisors, etc.).

Crucial for compiler work was Binford's idea to introduce two stacks (for integers and floating-point numbers) for numerical intermediate results. In 1969, Golden and E.C. Rosen began transforming the compiler to generate efficient code for numerical programs. Declarations were introduced to eliminate type queries at runtime; the data representations created, the number stacks, and the call sequences built on them for numerical subprograms collectively ensured that the new compiler generated code comparable to that generated by a FORTRAN compiler [FAT71].

An early version of this compiler ran in 1971 [GOE70] and was complemented by optimization parts by White, a project completed around the end of 1976. During this time, the compiler, as the second most important project alongside improving the arithmetic base, underwent significant changes and revisions.

To conclude remarks on the developments of this former project, the structure of arithmetic for arbitrarily large (or precise) integers, adopted from D.E. Knuth [KN69], must be mentioned.

Until 1967, the compiler, like today in various LISP systems, was entirely written in LISP. It produced a list of assembler instructions, noted in a list form, briefly called LAP. Although White had revised this program in 1967 so that the LAP list could be stored externally and converted to machine code when needed [WHI67], this format was still too cumbersome. Furthermore, the code delivered by the compiler was certainly improvable.

The beginning was made in 1967 by W. Diffie, who rewrote the compiler in assembler and incorporated tables instead of cumbersome list operations [GOE70]. Parallel to the work on the compiler for fast arithmetic, considerable effort was devoted to optimizing the object code, meticulously checking for all error possibilities and rectifying any deficiencies found. Golden, White, and Rosen developed a new format for relocatable object modules (i.e., ma-

chine programs with some load-time address connections) based on a scheme for dynamic arrays designed by White. When the loading procedure using this format (developed by G.L. Steele and White) became operational between 1972 and 1973, processing times were significantly reduced. White [WHI77] reports a reduction in the loading time of the entire MACSYMA system from one hour to 2 minutes. Shortly thereafter, the new loading procedure was supplemented by the ability to automatically load programs only when needed.

Considering the time savings, one can easily imagine the many more application possibilities these changes brought about. While changes and expansions made primarily for the MACSYMA system mainly concerned the arithmetic base, demands from AI program systems hinted at desirable extensions of control structures.

As early as 1969, parallel to solutions offered by the MULTICS operating system, Sussman had demanded similar features for LISP: interruption mechanisms such as control characters, alarm clocks, inter-process communication, and exception conditions ([WHI77], p.2). About the same time as BBN-LISP (see Section 5.2), corresponding tools were implemented. Over time, demands became more precise, and dangers in the routines used for interruption handling were recognized (protection of embedded program systems in LISP from unwanted, LISP-specific interruptions, etc.) The most recent revision was carried out by Steele at the end of 1973.

In connection with the implementation of CONNIVER by Sussman, the need for non-local jumps arose. Since this desire was limited to tools for interrupting an ongoing calculation, White proposed language elements for global exits in 1972. Using the CATCH function, a nesting level is now marked, allowing a return from any point within that level (THROW).

To carefully separate the hierarchies of nested program systems (e.g., 1. LISP, 2. CONNIVER in LISP, 3. Problem program in CONNIVER), White and Sussman transformed the object list (list of all atom symbols) into a replaceable field (OBARRAY) in 1971. This eliminated conflict possibilities between names at different program levels. Further changes associated with this allowed users to completely alter fundamental actions of the LISP system, such as the processing cycle, to adapt to their requirements [WHI77].

Inspired by extensions in other LISP systems, MacLISP, as the language version has been called since around 1970, underwent further changes. Influences from STANFORD LISP1.6 [QA72a] on the design of the program system for dynamic changes in input and output sources are noteworthy ([WHI77], p.4), which were later redesigned following the MULTICS model (NEWIO) around 1975 (Steele).

InterLISP also provided suggestions for improvement by introducing a BREAK state in case of errors and for programmed interruption, allowing evaluation dialogues to be processed as in normal system interaction (at the main level). Programs comparable to InterLISP test and editing aids (which are also noted in LISP) were gradually created and practically accessible to every user.

Until 1971, each program worked in a rigid address space predetermined by the machine. The division of this space among different data types was not

changeable. During the development of MICROPLANNER, Sussman demanded possibilities for *intelligent dynamic memory management*, and in 1971, White introduced extensible subspaces. With the help of programmable parameters for garbage collection, the user could introduce upper limits for the subspaces and specify how much free space must be available at a minimum.

In 1972, this was solved by the GC Daemon—a program that is called immediately after each garbage collection and can monitor and control the use of memory areas by appropriately changing the parameters. Memory management was significantly improved by the consistent page orientation, a concept adopted from InterLISP. Plans for this were drafted in 1973 by White, Steele, and S. Macracis, and Steele programmed the corresponding routines in 1974. The use of type-pure pages not only achieved improved behavior during memory expansion but also made it possible to release pages containing immutable LISP data or programs for multiple use.

When multiple users are working simultaneously with, for example, MACSYMA, they do not each need to load the MACSYMA programs individually. They are only in memory once and protected from changes by a write lock. This significantly reduces the frequency of cases where pages need to be reloaded (page faults). Additionally, the garbage collector can ignore these pages since there are no pointers from them to the dynamically changing parts.

Around 1970-71, Martin separated from the MACSYMA group and began his work on automatic programming. Since he was now heavily dependent on the MULTICS system, he wished for a LISP system that is fully compatible with MacLISP running on the pdp-10. The MULTICS version was initiated around the end of 1971 by D.P. Reed and completed in 1973, with D.A. Moon among others being involved in the final stages.

Similar situations had occurred before. Whenever an employee left MIT, they wished for the old working environment. However, the attempts to export the MacLISP system failed because the operating systems on the pdp-10 were different. In many cases, the computer was also too small, as indicated by White [WHI77], who mentioned that the compiler alone needed 65K words in 197?.

In the summer of 1973, a new attempt was made to make MacLISP available at least within the Time-Sharing System TOPS-10 provided by the manufacturer of the pdp-10. This project was successfully completed in collaboration with employees of the Worcester Polytechnical Computation Centers and the Computer Science Department of Carnegie Mellon University.

Even integration into the operating system TENEX used for InterLISP was successful in 1971, although no users were found.

Thus, MacLISP is currently found in three significantly different environments: In the AI Laboratory, it runs on the pdp-10 under the ITS operating system (today: DEC-10), in MATHLAB (MACSYMA group), and outside MIT, it runs on Tops-10 (a slightly modified version operates in the Time-Sharing System of the AI Laboratory at Stanford University) also on the pdp-10. Additionally, it is available on the Honeywell H6180 under the MULTICS system.

MacLISP has a large number of implemented basic functions (over 220), among which the diversity of arithmetic functions stands out. Through a con-

siderable number of functions, the user can also intervene in the interpreter's work, enter into direct communication with the base operating system, and utilize the extensive array of peripheral devices [MOO74].

The operating systems that manage virtual memory for the user, offering almost unlimited space, introduce slowing effects due to the many possibilities they provide.

The Time-Sharing environment is once again unfavorable for a calculation that requires disproportionately large main memory space in many respects. This prompted the scientists at MIT to plan and build a machine that processes LISP at the machine level and provides a larger address space than the pdp-10. This LISP machine, roughly equivalent to MacLISP, was operational in 1977 [GREL74, BAW77].

In addition, MacLISP is also being further developed conventionally. For the new generation of DEC computers, specifically the VAX11/780, a LISP system is designed that builds on the experiences gained with MacLISP. To express the new ideas, this emerging system (in 1978) is named NIL, i.e., a new implementation of LISP. As heard, this system is supposed to be somewhat machine-independent to simplify transportation to other facilities and modeled after the LISP dialect of the Greenblatt LISP machine.

4.2 InterLISP

At BBN, sufficient experience had just been gained with software paging on the pdp-1 (see Section 4.8) when the new SDS940 machine arrived in mid-1966. This machine was not only three times faster than the pdp-1 but also had 24 bits per word and a physical memory of 64K words, providing more space.

The computer was built for time-sharing tasks and offered limited support for a small virtual memory. Bobrow and Murphy decided, once again, to establish a software-paging system beyond the capabilities given by the hardware, similar to what was done with the pdp-1. This virtual address space was to be 2048K words, requiring 21 bits.

Otherwise, the new LISP system was modeled after the old one on the pdp-1. The division of atoms into value cell, function cell, P-list, and P-name pointers was retained, and the "smart" CONS, designed to prevent scattering across pages, remained, and so on.

The compatible compiler (which could translate any function without SPECIAL declarations) was also adopted, with cooperation again from Deutsch and Bobrow. (Deutsch also advised the implementation group, which included L. Darley, in the design of the software pager.)

However, this LISP system acquired its distinctive features through user aids, which were not mere appendages in LISP form, as in other systems of that time, but were deeply integrated into the system (although parts were certainly noted in LISP). Responsible for this innovation was W. Teitelman. As mentioned earlier, he had some involvement in the development of pdp-6-LISP at MIT. He had been engaged in programming technology early on and designed program systems for debugging and monitoring LISP programs [TE65a]. Through the

development of the formatted list processing system FLIP [TE65b], he aimed to simplify and popularize list processing.

After Teitelman completed a system for interactive program development [TE66] in September 1966, he moved to BBN. There, he played a significant role in the design and programming of the emerging system for the SD940. In addition to working on the interpreter and, most importantly, on error-handling programs, logging functions, and interruption systems, Teitelman built several large subsystems to significantly facilitate the work of LISP users.

Among these, the structure-oriented editor deserves special mention. The idea came from Deutsch, who probably also provided an initial version. However, Teitelman significantly reprogrammed it when the LISP system became operational at the end of 1966. Teitelman later enhanced user aids through an improved FLIP [TE67] and the subsystem "DWIM" (Do What I Mean) for automatic error correction. This interactive system, especially useful for complex errors, was incorporated into the BBN-940-LISP in 1968 and served mainly for handling typos and errors.

By mid-1967, the system was handed over to users on the SDS940, and Teitelman authored a manual [BO67].

The next stage began with the purchase of a pdp-10 by BBN. Once again, starting from SDS940 LISP, a backward-compatible system was designed. Besides Bobrow and Murphy, A.K. Hartley and Teitelman were now involved; Hartley played a major role in the implementation, with Murphy's assistance. They not only transferred the programs but also modified the compiler to generate machine code for the pdp-10. The new system was named BBN-LISP [TE71].

The basis of the system was (and is) the time-sharing system TENEX [BO72], based on (slightly modified) hardware paging of the pdp-10, managing a virtual memory of 256k words. Unlike previous systems, now one machine word sufficed to hold an entire LISP cell. Although the theoretically available memory decreased (to 512K pointers), more was practically accessible since the SDS940 had operated with orders of magnitude of 96K.

Later, Wegbreit and J.W. Goodwin added a type of separate virtual memory for compiled programs—the Swapper. With this addition, the space for programs in this form increased to 128 million words.

A new aspect with the transition to the pdp-10 was that a considerable number of LISP programs already existed. Many of these were explicit system programs that needed to be incorporated into the new system. To avoid the overwhelming task of rewriting, Goodwin developed a LISP program, TRANSOR, to execute this conversion. TRANSOR was later expanded so that it could convert LISP programs from other LISP dialects, such as LISP1.5, STANFORD LISP1.6, etc., into InterLISP.

Now, the LISP system provided such a foundation that the user aids were further developed. The DWIM subsystem was evolved, and systems like Programmer's Assistant [TE72b] and CLISP [TE76], i.e., Conversational LISP, were included in the system. In 1971, Bobrow completed a block compiler that translated entire LISP programs more favorably than the normal compiler. Of-

ten, dealing with systems of LISP programs that are only interdependent and are externally accessed through a main function, the loose concatenation by the normal LISP compiler and the management of functions through a ring buffer in limited space brought about issues. (The higher-level calling function is displaced by the called function.) By introducing faster and tighter connections between functions in the function block and by the ability to arrange global variables only once, namely upon entry into the block, in the stack memory, significant time is saved. Even the execution of a single recursive function is simplified when translated by the block compiler ([TE71], p.18, 17).

BBN-LISP soon became a very well-known and sought-after system. Especially the numerous testing and utility aids were very attractive to implementers of other LISP systems. Therefore, it is not surprising that attempts were soon made to integrate these subsystems into other existing LISP systems. A notable venture in this direction is represented by UCI LISP [BOB73]. The construction of this system began in 1971 by revising a STANFORD LISP 1.6 system [QA72a] and placing it in a new environment from the perspective of BBN-LISP. Extensive work was required to make the programs reentrant; thus, the division into a shareable program part of 15K words and a user-specific part of 8K words became possible. Since the leader of this project was a brother of D.G. Bobrow (Rusty Robert J. Bobrow), a good connection to BBN was ensured — Bobrow, Teitelman, Goodwin, and Hartley provided guidance through many pieces of advice. Diffie, from Stanford, ensured that the programming team (R.J. Bobrow, R.R. Burton, J.M. Jacobs, and D. Lewis) had no unresolved questions regarding the LISP 1.6 system.

UCI-LISP is now characterized by efficient memory usage thanks to reentrant code, simple ways to continuously reallocate memory areas, expansion of LISP functions, and the inclusion of interactive error detection and correction programs from BBN. Other improvements affected input and output programs.

When studying the completion report of UCI-LISP, it is interesting to note that many parts of STANFORD LISP 1.6 were corrected ([BOB73], p. 0.8); such errors were found in the compiler and LAP (where there are always rarely used special cases that turn out to be faulty) and even in the garbage collector. And this in a frequently used program after years of runtime!

During the subsequent period, BBN-LISP also underwent changes. In 1972, Teitelman and D. Bobrow joined the XEROX Palo Alto Research Center, where intensive research on office automation based on natural language processing began. As the LISP system was now maintained by two institutions, the name was changed to InterLISP in 1973 [TE74].

In 1973, D. Bobrow and Wegbreit proposed an implementation technique for working with arbitrary variable environments [BOB73c,d]. This was intended to solve, on the one hand, the problem of global variables only partially addressed in the stack binding of InterLISP (the FUNARG problem), and on the other hand, research in the field of artificial intelligence had led to the view that interchangeable binding environments (so-called "worlds") were essential programming tools (see pp. 149ff.).

BBN-LISP and its successor, InterLISP, like LISP 1.6 and MacLISP, do not

have an association list like LISP 1.5. The reason for this decision was the observation that in a virtual system, the association list is relatively often (as a list structure!) at least partially outsourced to external storage media. The stack memory, however, is almost constantly resident in its most important parts due to the frequent operations performed with it. Retrieving values via atoms, as is common in LISP 1.6, is undesirable for the same reason as retrieving via the A-list: many page swapping operations would result.

The solution, already developed for the pdp-1-LISP in 1965, is to store pairs of atoms and values in the stack memory. This essentially "linearizes" the A-list. The advantages of this decision include, as intended, faster access to values compared to the A-list (when no page swapping is required, the LISP 1.6 scheme proves superior), as well as greater simplicity in combining variable access from compiled functions and interpreters (hence the compatible compiler).

The stack memory solution allows correct handling of global variables in simple cases (upward FUNARG): if the functional argument appears in an outer binding scope and uses the variables globally there, when working with this functional argument in inner variable environments, only the irrelevant new variable bindings need to be skipped.

However, providing variable environments correctly becomes complicated when such a functional argument is delivered as a value to an outer binding environment (downward FUNARG) or when using the functional argument itself results in nested environments. Then, the linear arrangement or the stack memory principle is no longer sufficient; a list-like structure must be used. To use closed stack memory pieces with the associated processing speed advantage in the linear intermediate sections (where no functional arguments occur), Bobrow and Wegbreit proposed the Spaghetti Stack.

According to the theoretically developed model, the Spaghetti Stack was implemented (by Hartley) and was available around 1975. In the meantime, Wegbreit and Goodwin completed the Swapper for compiled programs (1974).

The maintenance of the InterLISP system lay with Teitelman, Hartley, Lewis, and L.M. Masinter after 1975. Teitelman was responsible for user tools, i.e., he supervised the Prettyprinter, Editor, Break and Trace programs, the Advisor, Printstructure, and the subsystems DWIM, CLISP, Programmer's Assistant, etc. In 1977, he reported on further development of the user dialogue support in InterLISP [TE77], which allowed users to master multiple complex program environments simultaneously and pursue various tasks through menus and windows [12].

Hartley was responsible for the actual system programs, i.e., for the interpreter, garbage collector, compiler, Spaghetti Stack, as well as all hand-coded functions in machine language. Lewis's area of responsibility included all questions related to input and output and the external environment (i.e., issues related to the operating system). Masinter managed the pattern match compiler,

[12]Menu is a term from digital graphics. By specifying an ordered set of operators (the menu), the user can manipulate objects on the screen by pointing to the desired operation. This eliminates the need to resort to formulation tools that are inconvenient and difficult to oversee.

the program package for records (and user-defined data types), and MASTERSCOPE.

Similar to MacLISP developers, the InterLISP designers also believed in a future of the system at the hardware level in a microprogrammed computer. While BBN started with a pdp-11, XEROX tried to design a special small computer. The basic ideas for this were already published by Deutsch in 1973 [DEU73]. The result of this effort can be inferred for BBN from a progress report in 1977 [ASH77]. For a long time, not much was known about XEROX's progress, because, as Fredkin put it: "XEROX has more secrets to protect than the entire U.S. government" [FRE77]. Then, in 197?, Dolphin appeared on the market... Dandelion, Star,...

Unlike MacLISP, since 1972, various parts of the world have seen the emergence of additional InterLISP implementations. Implementations are known for IBM/360, Siemens 4004, ICL-4, CDC3300, Burroughs 6700, and Fujitso M160 and M190 [13]. This transfer began with the initiative of the Artificial Intelligence Working Group under Sandevall in Uppsala. Jaak Urmi designed a smaller experimental version of BBN-LISP for the Siemens 305 mainframe in 1970 and gained some experience. When the university decided to purchase an IBM/360 in 1971, contacts were made with BBN to adopt InterLISP. This was successfully accomplished. Thus, in the early summer of 1972, Urmi, B. Jansson, and M. Anderson began programming. Since significant parts of InterLISP were written in LISP, the main task was to implement the page management system.

After just over a year, an experimental version (estimated by Urmi to be 3 to 5 person-years) ran on OS/MVT. When the system was reasonably stable, requests for further transfers arose. Therefore, in 1975–76, InterLISP was adopted from Uppsala to Zurich on an IBM/370 under the CMS operating system.

Around the same time, inquiries came from the UK and West Germany regarding the adoption of InterLISP on IBM-compatible Siemens 4004 and ICL-4 computers. In both cases, it was believed that the work would only take a few weeks, but it turned out that small incompatibilities in the operating systems brought unpleasant problems.

The transfer to Siemens 4004 was completed with Urmi's assistance within a few months. From the end of 1977, Siemens' research lab managed this system independently. The ICL-4 transfer is said to have been completed in Edinburgh at the end of 1977.

From Uppsala, InterLISP was transferred to Japan. Initially, IBM/370.158 and 138 were used as target systems. Later, the Japanese Fujitso M160 and M190 computers were used (February 1978).

Little is known about the other InterLISP implementations. The B6700 adoption in Los Angeles seems not to have been completed.

[13] As heard, from 197?, there was no pdp-10 in Palo Alto. Instead, LISP and TENEX enthusiasts had to create an emulator on the available computer (see [TE77]).

4.3 STANFORD LISP 1.6

When the first version of LISP 1.6 was reasonably stable at MIT in 1967, the program was transferred to Stanford, where a pdp-6 was also in operation. This system was now adapted to the specific requirements, especially the local time-sharing system. Allen and L. Quam revised the system and, among other things, improved the input procedure by controlling it with a fast (superfast) table-driven scanner that could also be easily modified by users [QA72a]. The main work here was done by Quam, while Allen built a new memory allocation system that allowed dynamic changes in memory segment sizes. Additionally, he developed the ALVINE editor.

Diffie, who had been maintaining this system since around 1970, improved it based on his experiences with the compiler ([QA72a], App.F). However, "the LISP 1.6 compiler does not seem to have undergone as complete verification and error checking as the MACLISP compiler, as its recommendation is unreliability" ([WHI77], p.7). Developers of UCI-LISP, who used STANFORD LISP 1.6 as a base, also reported errors in garbage collection.

Earlier than in other LISP systems, Stanford recognized the need to make the relationship between physical and logical input devices much looser than was customary up to that point.

The typical situation in a LISP system was that the user had two main input and output units: input, usually a card reader or teletype, and output, usually a printer or teletype. For technical reasons, additional input and output units were assigned, but they were beyond the actual access of the user (for system dumps, etc.). Later, devices with libraries were added, from which a library was loaded in chunks or only the desired function.

LISP 1.6 then took the first step in the free interchangeability of external devices in 1967. For each device, there were specific control characters that activated the device. Especially for DEC tapes (small magnetic tapes fixed in two-axis cassettes on which intermediate results were usually stored), a series of functions had been developed that connected the file directory to the LISP system, detached it, and controlled input or output. The display was completely separate from the system of devices [PDP6c].

STANFORD LISP 1.6 unified this patchwork into a concept of logical files. Regardless of the device, whether a subfile or similar, it can be declared as a new input channel with the INPUT function, which can then be declared or made passive with INC as the current input source. The same applies to OUTPUT and OUTC regarding output. The user was thus faced with a logically unlimited variety of input and output options from which he could dynamically choose during the runtime of his programs (see [Kur78]).

STANFORD LISP 1.6 is now one of the most widely used LISP systems. Wherever pdp-10 systems are used with LISP (with the few exceptions where InterLISP or MacLISP is available), this system is used. It is also distributed through the DECUS user community.

The further development to UCI-LISP has been discussed in the section on InterLISP. This system was further developed at Rutgers University in 1977–78.

However, around 1980, LISP 1.6 was hardly used at Stanford itself. But it was exported with the pdp-10. R.P. Gabriel had brought MacLISP to Stanford in 198?, integrating it into the local WAITS operating system.

4.4 Other LISP implementations in the USA

In early 1967, Stanford University learned about Jan Kent's work at Waterloo. Since they were in the process of installing an IBM/360.91 at that time, they were very interested in having a functional LISP system for this facility, similar to Waterloo. Kent came to Stanford and brought the completed program parts with him[14]. Together with R.I. Berns, he completed and finalized the system over the course of the year. It now included, in addition to the interpreter, which closely resembled that of Waterloo, a compiler (adopted from IBM).

When Kent left Stanford at the end of the year, this LISP system, now generally called STANFORD LISP/360, was considered a relatively powerful and fast system. Its purpose was to enable the processing of extremely large programs.

In the first year, alongside Kent and Berns, Hearn, F.W.Blair, and J.H.Griesmer[15] also contributed to the implementation. However, since 1968, only Berns has been responsible for the further development in Stanford. Blair and Griesmer had previously developed a local implementation (in Yorktown Heights) and continued working with LISP at IBM [BLA78b], and Hearn later initiated the further development of STANFORD LISP/360 for use as the base system for his REDUCE formula manipulation system [HEA73] in Salt Lake City.

Characteristic of STANFORD LISP/360 is that this system did not build on the experiences available at MIT or BBN at the time but was almost entirely compatible with LISP 1.5. When studying the programs, it becomes apparent that the code largely corresponds to the description of LISP interpretation from the LISP 1.5 Manual [MCC62c] with the LISP-M language, which was meant only "pedagogically." For example, when querying the function type, separate calls to the GET subroutine (which retrieves the associated meaning) are triggered for each option, instead of working through the P-list at once.

Overall, the work of the interpreter appears poorly solved, while many excellent detailed solutions were found for list processing. Among them is the entire stack memory work (which is efficient) and the implementation of the basic functions CONS, which does without comparisons regarding the end of the free memory list. Instead, an interruption is triggered, and garbage collection is invoked—this, in its compactness and brevity, is an excellent example of the harmonious design of fundamental decisions in the system.

From today's perspective, however, some of them are questionable: pointers should, in any case, include their type in the IBM/360 and not just the ob-

[14]There is information that the "bringing" of the programs from Waterloo to Stanford may not have gone entirely smoothly...

[15]the latter indirectly, as they were working on a similar system in Yorktown Heights

ject they point to; garbage collection makes unnecessarily frequent use of stack memory [KUR76b]. Nevertheless, detailed criticism will be avoided here.

Since 1967, STANFORD LISP/360 has not undergone any serious revisions but has only grown through the addition of new functions and the extension of existing capabilities. In addition to maintaining the system [LISP], Berns, in Stanford, embedded it into the Stanford Time-Sharing System on IBM/360.67 [BERN71].

In Utah, as well as in the majority of installations outside the USA, the system was primarily used to support REDUCE. For these purposes, the LISP system did not meet all expectations. To address difficulties with memory dumps and the distribution of memory segments, a lot of work was done [KAY75]. The significant innovations from Salt Lake City, however, are the arbitrary-sized integers that were integrated into the system of simple data types. The programs for this were taken almost unchanged from STANFORD LISP 1.6 (where they were programmed based on Collin's formula manipulation system SAC [COL71]). Further minor improvements and the integration of STANFORD LISP/360 into the Time Sharing System CP/CMS were carried out in France by A. Lux at the University of Grenoble [LUX77].

STANFORD LISP/360 was likely the most widely used LISP system in the world around 1980. This is due, on the one hand, to the fact that this system was practically the only one distributed for IBM/360 installations, and on the other hand, the almost free distribution in conjunction with REDUCE contributed significantly to its popularity.

In connection with REDUCE, there is another aspect that has not been without an impact on the history of LISP: In 1964, Hearn [HEA64] began working on physics problems with LISP programs for formula manipulation. The successes encouraged him to systematically approach the field of symbolic processing. He began building the REDUCE system in the LISP 1.5 system on the IBM/7090 at Stanford University. Soon, the first interested parties appeared (Hearn himself is a physicist), and the system could be used on various IBM/7090 installations and on the Q-32 of the System Development Corporation due to the compatibility of LISP implementations. Between 1964 and 1965, he brought the earlier version of his system to England and West Germany. However, shortly after the first manual was written in 1967 [HEA67], the situation changed: The pdp-6 was installed at Stanford, and LISP 1.6 was available. This system differed significantly from LISP 1.5 in some important aspects. The situation persisted when STANFORD LISP/360 became another, essentially correct, LISP 1.5 that could be used.

Hearn took the path of unifying the starting point for REDUCE with a preprocessor in the various base systems and building the actual REDUCE on top of this base system. This made it possible for the same LISP program system to be usable on two quite different computers and different LISP versions [HEA67].

The discussion with the newly emerging LISP systems at BBN (BBN-SDS940-LISP) and the University of Texas in Austin (UT-LISP) led to the idea of making all these systems compatible with the help of a preprocessor, i.e., creating

a standardized LISP environment. Hearn formulated his ideas in early 1969 [HEA69]. The basis of STANDARD LISP, as he called this unified foundation, should be LISP 1.5. Of the 139 basic functions in LISP 1.5 [MCC62c], 67 were to remain completely unchanged. These were primarily functions for list manipulation, functionals, the PROG feature, and the majority of arithmetic and P-list functions. Eight functions, including ERRORSET and SETQ, were assigned a slightly different meaning (SETQ should also be usable for free variables instead of CSETQ). In six subject areas, STANDARD LISP took into account developments since 1962 and went beyond LISP 1.5:

1. In the treatment of free variables and constants. In this field, there were significant differences between different LISP versions in 1968-69. Since STANDARD LISP, in Hearn's opinion, could not adopt the more modern solutions of LISP 1.6 or BBN LISP, limitations had to be imposed. According to Hearn, three cases were manageable: Constants, which would be declared at the beginning along with their values and could then be replaced by the respective values. Global variables that do not appear in any variable list of any LAMBDA or PROG. In this case, no conflict can arise between A-list implementation and stack memory implementation—a SPECIAL declaration is possible, which sets up a value cell. For interpretative evaluation, extra access or storage functions must be used (GTS, PTS from GET SPECIAL, PUT SPECIAL).

Finally, variables that are bound at some point and are free at other points had to be considered; Hearn called them extended variables. If such variables are used, according to the solution in STANDARD LISP, the entirety of the affected functions must either be compiled or be available for interpretation.

2. Functional arguments. To distinguish between functional arguments that refer to the environment at binding time and those that do not, the solution from LISP 1.6 was adopted. Accordingly, the simpler cases are handled with FUNCTION, i.e., practically quoted, while the others are marked with ∗FUNCTION. The occurring free variables must, of course, be declared.

3. Character handling. When reading character by character or when breaking down an atom name into its components (via EXPLODE), atoms with a name consisting of only one character are created. STANDARD LISP proposed a special distinction here between character atoms and ordinary (literal) atoms. Character atoms have no P-lists and are not related to the atom with the same name. Six functions for working with these character atoms were proposed (READCH, PRINC, EXPLODE, COMPRESS, LITER, and DIGIT), along with some for controlling the output (PRIN, OTLL, POS, SPACES). These probably would have been better placed in the next group.

4. File management. Compared to LISP 1.5, STANFORD LISP had to allow for changing the source of the input stream and the target for the output stream. No other possibilities for working with external files were considered. The proposed functions (OPEN, WRS, RDS, CLOSE) allowed declaration, switching to the new file, and closing.

5. Functional and macro definition. Strangely, STANDARD LISP remained somewhat undecided on this aspect, even though some new ideas were already present in 1969. Impressed by the continuous compilation of Q-32-LISP, defi-

nition and compilation should, in principle, coincide. With DEFINE (parallel to DEFEXPR), EXPR should be defined. For Macros, the function MACRO was intended; otherwise, they followed the ideas of Harts [HART63b], mediated through C.Weissman [WEI67]. Additionally, every function definition should be reachable via GETD.

6. Arrays. The development in this area was characterized at that time by the introduction of array types whose elements (for efficiency reasons) could not be valid LISP objects. STANDARD LISP proposed the strict limitation of the functions SETEL and GETEL for storing or accessing array elements.

In addition, there is a mixed sequence of limitations compared to LISP 1.5, determined by the need to find the lowest common denominator of all currently existing LISP systems. Atom symbols with 24 characters, only five arguments for functions, the absence of LABEL, processing with EVALQUOTE, etc., are characteristic examples of these restrictions.

As commendable as the approach to STANDARD LISP was, as the success of standardization efforts for LISP could have significantly influenced further language development, the somewhat unsustainable combination of new ideas and restricted parameters from contemporary LISP implementations could achieve little more than establishing a private language standard. Moreover, unfortunately, the LISP community did not yet see the necessity of standardization.

Nevertheless, the development of preprocessors for STANFORD LISP 1.6, STANFORD LISP/360, and CDC 6000 LISP 1.5 (UT-LISP) laid the groundwork for a considerable degree of machine independence for the REDUCE formula manipulation system. A significant drawback of the preprocessor approach became apparent when RLISP, the ALGOL-like formulation language for REDUCE, was understood as a LISP extension equivalent to MLISP or CGOL, and attempts were made to write programs that should process STANDARD LISP programs.

The STANDARD LISP model assumed that there was a "host system" on the given computer, overlaid by the preprocessor with STANDARD LISP. This led to the implementation of the same STANDARD LISP functions being different on various machines (or in different LISP systems). For a machine-independent compiler, which, according to Hearn's ideas, would have to be written in RLISP for STANDARD LISP, the task would be complicated by system-dependent source programs. The conclusion from this situation for Hearn and his colleagues was that STANDARD LISP needed a precise language description, specifying the data structures and operation of the basic functions precisely [MAR77].

The new report for STANDARD LISP contained these specifications. The intention was that an existing LISP system should now be modified according to the standard. The report itself should only be the first step. The authors intended, similar to what J.S. Moore had done for InterLISP [MOO76], to present a defining description at the implementation level.

Nobody expected in 1977 that this report could actually introduce a language standard comparable to the standardization of FORTRAN [FOR66] or COBOL [COB74]. The development of the leading LISP systems had progressed so far that standardizing LISP seemed hopeless. Much would have been achieved if

the different implementations of these systems were somewhat compatible. In 1969, the situation was more favorable.

However, within the circle of those interested in formula manipulation, besides MacLISP and InterLISP, only STANDARD LISP would have been used, this would have been a remarkable success. This did not happen. STANDARD LISP remained one dialect among several, developed by friends and implemented by them on various machines. Implementations became known on IBM/360/370, pdp-10 (DEC-10, DEC-20), B6700, B1726, and UNIVAC 1108.

The data types in STANDARD LISP correspond to the spectrum commonly implemented. The names of atom symbols are again limited to 24 characters—a decision that is difficult to understand.

The function types are described using the terminology established by InterLISP: Independently, there are the features EVAL or NOEVAL for functions that receive their arguments evaluated or unevaluated, and SPREAD or NOSPREAD for functions to which the arguments are distributed to parameters or assigned as a list to a parameter.

The described functions—there are about 115—constitute the usual basic set. It is noticeable that certain conventional ideas from LISP 1.5 were adopted. An example of this is the so-called flags in the P-lists, whose relationships to ordinary properties are described rather vaguely. This is because in LISP 1.6 flags no longer played a special role, and proper properties with the values NIL and T are used, while in STANFORD LISP/360, indicators without properties are used. Sandewall's functions DE, DF, and DM were accepted for function definition. All functions for character manipulation were rigorously excluded from input and output functions this time. This makes sense for COMPRESS and EXPLODE since the character list can be imagined as the source or target (see p.62).

Much thought has been given to variable types. To distinguish the global variables, called Extended Variables in 1969, the variable type of Fluid Variables was introduced. These are bound somewhere and are globally used in subfunctions. When a new binding occurs, the old one must be saved. Only the currently active one is accessible. Cooperation between interpreted and compiled functions is assumed to be secure.

Work on the portable LISP compiler, which began with the old STANFORD LISP/360 compiler and went through several intermediate stages (initially planned optimizations of the source code were first eliminated), was completed by 1983 [HEA83].

STANFORD LISP/360 was not the only LISP system for IBM/360 or /370 computers. Many machines acquired by universities and research institutes were sometimes operated with different operating systems. In the first LISP Bulletin, published in April 1969 in the SIGPLAN Notices [BO69], alone, three LISP systems for these installations are mentioned[16]. One was implemented in early 1968 at SDC in Santa Monica as a successor to Q-32-LISP for the

[16]Curiously (one of the many inconsistencies in the assessable literature), LISP/360 is not mentioned for Waterloo or Stanford.

ADEPT operating system [LI69] on an IBM/360.50 [BAR68]. Another was at the Rensselaer Polytechnic Institute in Troy (NY.), and finally, one was for the IBM/360.91 at the IBM Research Institute in Yorktown Heights, running under various operating systems: in OS, in OS with TSS, and in the CP/CMS system.

At IBM, the offering of various LISP systems that had been implemented there had long been avoided as either commercially uninteresting or causing too much trouble. Consequently, little is known about the work that Blair and Griesmer primarily carried out over the years.

According to Blair, it was possible to use an MIT LISP 1.5 system in Yorktown Heights as early as 1961. This likely addressed problems of formula manipulation and other non-numeric tasks. However, the user base was very limited each year. Between 1965 and 1966, access to the LISP system on the IBM M44 seems to have been available, as reported by E.G. Wagner [WAG78]. It is unclear whether this was the system created by D. Wooldrige with HashCONS [BOB68b][p.152] or another system built by Edwards, perhaps with the assistance of Abrahams [BLA71]. Shortly thereafter, Blair and Griesmer were involved in the implementation of LISP 2 on the IBM/360. Since this project collapsed, they started their own implementation for this computer.

Characteristically for this IBM system is the orientation towards half-words. This saved space but, on the other hand, only provided a small address space. To make it sufficient, list structures were placed in this area as much as possible. Atoms were, similar to the pdp-1-LISP from BBN, broken down into parts, of which only a part was in the pointer area. From this part, pointers went to the bucket list of the object list and to the p-list.

The system included integers and floating-point numbers, character strings, and vectors. It offered a fairly extensive range of functions. The compiler was already developed to a significant extent by mid-1967, so that Kent could adopt it for STANFORD LISP/360 after a visit to Yorktown Heights.

Maurer [MAU73] reports on a LISP 1.4, in which his examples were processed with the help of Griesmer, and in which the larger report mentions the version LISP 1.15. It is not unlikely that the systems mentioned by J. Sammet [SAM69] without reference were IBM products LISP 1.17 or LISP 1.19.

Each respective system was considered an "experimental system," available to a small group of no more than five programmers for use.

Around 1970, this system was used to build the formula manipulation system SCRATCHPAD [GRI71b]. The technology used was generalized, and a descriptive language for metasystems [JE70] was developed. In the CP/CMS operating system, the meta-language LPL [BLA70a] could be used. Its level was roughly equivalent to MLISP. Since syntax processing was based on V. Pratt's ideas regarding Operator-Precedence Grammars, the system could process prefix operators extremely efficiently. For the user, this resulted in, for example,

A B C in (A(B(C)))

The "-" sign was used to switch to pure LISP syntax, and vector data objects used angle brackets. Like MLISP, LPL also included ALGOL-like cycle

constructions.

Around 1974, IBM employees considered whether to make a new attempt. The "Task Force" began its work, and decisions were made. In 1975, an entirely new implementation was started. This one was now to be based on double-word cells to have the entire address space available.

During the course of 1976, the essential work on the new LISP/370 system was carried out. White from MIT also contributed to this.

Characteristic of LISP/370 are the data structures organized as pointer complexes and, especially in input and output, quite unique functions.

In 1977, the LISP/370 system was announced and offered to users, and in 1978, it came to market as Installed User Program No. 5796-PKL [IBM78].

Another LISP system for IBM/360 machines was created in the early seventies at the University of Michigan in Ann Arbor by Bruce Wilcox and others [HAF74]. This system ran on the Michigan Terminal System and was designed closely following MacLISP. Therefore, it was initially called MacLISP/360. An improved version called MTS-LISP emerged around 1974. Since it was distributed free of charge, it is assumed to have been used externally [GOE76]. For example, it reached the Institute for the German Language in Mannheim.

In the first article on STANDARD LISP [HEA69], Hearn mentioned the LISP implementation conducted at the University of Texas in Austin. The history of this system is typical for the dissemination of LISP.

In a course, W.W. Bledsoe and S. Papert (from MIT) introduced the idea and results of the emerging discipline of Artificial Intelligence in Austin and inspired students to participate or assist in research in this field. This was in the fall of 1966. Programming was, of course, to be done in LISP, but there was no computer available for which a LISP version had been implemented. Bledsoe encouraged the students to start with the LISP implementation for the university's computer, a CDC6600 [MOR76].

They initiated a Graduate Student Project, which was launched in the fall by students J.B. Morris, R. Richardson, and D.C. Singleton. Later, an undergraduate student, R. Fredrickson, joined them. Together, they designed and programmed a LISP 1.5 system. This program was reportedly ready for use around the end of the spring semester in 1967.

The design work was not straightforward, as the machine architecture of the CDC6600 is quite different from other machines. Unfavorable for LISP is the enormous word length of 60 bits, against which there are 18-bit addresses. The students decided to use 6 bits per word as type pointers. This left 12 bits unused in the list structures! This is a bold decision, considering that no further variants were conceivable (for example, in atom structure) where LISP could have deviated for better memory utilization. However, the use of the third address part in the P-lists was obvious: you get the indicator, property, and list pointer in one word. The many bits can also be well-utilized for name cells, numbers, etc.

The CDC6600-LISP system apparently found external users immediately because there was a long-standing demand for it. As usual, there were still some errors; Morris and Singleton improved these and utilized the initial user

experiences [MOR68a]. In early 1968, after a closer examination of the system (LISP 1.55), it was even included in the software supported by the CDC6600 user community (CDC-VIM-6600 Users Group Software Library). By 1969, the system had been adopted by around 20 installations. Among others, it also reached the VR Poland in Warsaw.

Morris maintained the system until 1969. In the summer of 1969, E.M. Greenawalt took over the CDC6600-LISP system after him. He completely overhauled the system, added a compiler, and also wrote a LAP assembler program. Particularly noteworthy is the new interpretation scheme for λ-expressions, based on a variable binding technique that is different from both the A-list implementation in LISP 1.5 and the LISP 1.6 scheme, where values are accessible via a value cell arranged in the stack.

UT-LISP [GRE70], as the system is now called[17], uses a local push-down list, i.e., a stack represented as a list directly accessible from the atom (in the Car field). The current value is in the Cdr field. This makes value retrievals very fast to execute — however, managing through decentralized push-down lists does not seem to be very favorable.

Around 1980, UT-LISP (version 4.0) was one of the most widely used and functional systems [GRE75]. With approximately 220 functions, good interactive capabilities, and the freedom to use any input and output files, it is quite powerful. One feature that enhances these properties is the ability for the user to switch between different system configurations. In UT-LISP terminology, this service is called an "Overlay." When started with an empty LISP memory and a series of functions are read in, an working environment is eventually created, characterized by the internal availability of a system of functions. In this specific system configuration, programming work or normal problem-solving becomes possible. Since such a system configuration is only reachable through a long series of input operations, it makes sense to consider it as a whole unit and save it as a data element to external storage. For later system starts, only one input operation is needed — that of the Overlay.

In this regard, UT-LISP is not alone. STANFORD LISP/360, InterLISP, and DOS/ES LISP1.6 also allowed similar operations, but here, in operating system-dependent terminology, they are referred to as "Checkpoints." What makes UT-LISP unique is the ability to switch from one overlay to another, perform certain subtasks, and then re-enter the original system with the results. This approach can even be organized recursively [GRE73].

In 1973, Greenawalt spent time in Livermore at the Lawrence Radiation Laboratory and ensured the transfer of UT-LISP to the LTSS operating system running on the CDC7600. Development then stalled when Greenawalt left Austin in 1974. R.A. Amsler and J. Slocum then supervised the system. They consolidated all the last extensions and new elements made by Greenawalt [AMS75]. For CDC computers not equipped with the relatively rare operating system used in Austin, adaptation work must be carried out.

Further improvements to UT-LISP were also carried out in Lausanne, at the

[17] by the *University of Texas* at Austin

ISSCO Institute, under P. Hayes and N. Goldman [GOLM75].

For the UNIVAC 1100 computer family [IB78], a LISP system was developed only in 1969. Not much is known about the circumstances of this implementation. UNIVAC LISP is a strict LISP 1.5 system, meaning an association list is used for variable binding. The implementer, E. Norman, states that the SPECIAL variable implementation would not work ("it just doesn't work"), referring to the difficulties in achieving a correct FUNARG. His example, the composition function for two functional arguments, is both simple and natural and would indeed not be executed correctly in a simple LISP 1.6 implementation (such as DOS/ES LISP1.6).

```
(DE COMPOSE (F1 F2) (FUNCTION(LAMBDA(X) (F1(F2 X))))).
```

This is one of the simplest examples of a so-called downward FUNARG.

Regarding the atom symbols, one can see that UNIVAC-LISP, in its rigidity, was something of a precursor to SCHEME. Atoms were built as an aggregate of four list cells (2 words) and, in addition to the value, P-list, and name pointer, also contained the pointer for the hash bucket (list of equal hash codes). There was no function cell! Atoms could not be used simultaneously as variables and functions. It was precisely the designer's intention to make this impossible — in this, he could have referred to McCarthy [MCC63e]. Functions are treated like normal objects and assigned using the functions CSET or CSETQ.

Initially, UNIVAC LISP offered around 100 basic functions. The arithmetic functions are quite prominent, the functions for list processing correspond approximately to the normal spectrum, and input and output functions are also adequately represented. In the set of functionals, very unusual naming conventions are found (MAPC is called MAPCAR, MAPCAR is called INTO). The effects of treating the function definition as a normal value are evident in the set of functions: one cannot define FEXPRs, and the concept of different function types is completely dissolved. The "special forms" like PROG, COND, etc., are apparently "built-in," and new ones cannot be added. UNIVAC LISP can thus be appreciated as a very modern LISP dialect. The departure from the notion that special forms are functions, taken up by other authors much later, can again be pointed to SCHEME.

In UNIVAC LISP, memory is divided into pages in which data is stored in a type-specific manner. A distinction is made between numbers, atoms (literal atoms), compiled programs, and some special types in addition to the list cells [NOR69b]. The page division has the advantage that the user does not have to pre-divide the memory. The pages of different types can be arranged arbitrarily. All types are covered by garbage collection. However, it is not compacted. Overall, the memory allocation seems very successful (it resembles the schemes of InterLISP and MacLISP (BIBOP)) and capable of extensions (software paging).

UNIVAC LISP has been adopted by the University of Maryland, among others. Work on the system continued there. Around 1973–74, R.L. Kirby created a compatible version for the pdp-11 [KIR75]. It now includes arrays but suffers overall from the disharmonious specifications of the model.

The pdp-11-LISP from Harvard University has also been very widely used. It was programmed in 1974–75 by F. Howard at the HRSTS Science Center in Lynnfield [HOW75]. Also for the pdp-11, there is the system by B. Webb, developed at the Bell Telephone Laboratory. All three run in the widely used UNIX operating system [RIC74].

5 LISP in Europe

5.1 United Kingdom

In the United Kingdom, the reception of LISP outside the USA began.

Around 1960, a loose circle of scientists and friends gathered at Christopher Strachey's for tea, discussing new developments in the field of computer science. When McCarthy's article appeared in the Communications of ACM in March 1960, attention was drawn to it, and people started engaging with it. Sometime shortly thereafter, M. Woodger of the Royal Radar Establishment in Malvern, Worcester, proposed implementing this interesting language to D.R. Jenkins.

Jenkins was interested, and there was some experience in language implementation at his institute. They had developed a simple algebraic language and written a compiler. P.W. Woodward, who had been involved in this project, was willing to support Jenkins. So, based on McCarthy's article "Recursive functions..." [MCC60c], they started implementing LISP.

At that time, the Royal Radar Establishment had the TREAC, a not-too-large computer equipped with a drum. It quickly became apparent that the small main memory was limiting. Due to the many drum transfers, LISP interpretation was quite slow. Unfortunately, in an article about LISP [WOW61], the implementors provided very little detail about their work, stating that it was too recent to address detailed experiences of real problems.

Nevertheless, the publication included a demonstration of basic arithmetic operations without relying on LISP base functions. A speed indication was also given: the system took about 5 seconds to calculate the differential quotient of x with respect to x (i.e., dx/dx).

The LISP system on TREAC served as an experience and testing tool. Shortly thereafter (before October 1961), Strachey's colleagues implemented LISP on an EMIDEC2400 [McC6?] [p. 94]. Later, with the participation of J.M. Foster, list processing was continued independently of LISP.

In the early sixties, list processing became relatively popular in the UK. Several systems were developed that were more or less different from LISP [FOX66, WILK65]. Among them, the language WISP deserves special mention [WILK64].

LISP became relevant when the first machines of the IBM 7090 series were delivered to the UK. It seems that the installation of an IBM 7090 at the Imperial College in late 1964 was the triggering event.

For this installation, Hearn, who was in England from September 1964 to August, introduced the LISP 1.55 system ([MCC62c], Appendix I, [CAM66]).

The motivation was the needs of physicists at the Imperial College and other universities or institutes in the London area, as Hearn had brought his formula manipulation system. The Imperial College was to serve as the central computing center. Hearn also promoted LISP during his stay at the Rutherford Laboratory in Didcot.

The system was developed until August 1966 to meet certain specific needs of physical research and was used by physicists at the Clarendon Laboratory in Oxford under the name LISP 1.6 on the London computer. However, a less specialized variant was used at the Imperial College [CAM66].

Although there were not many LISP programmers in early 1967, the application areas were relatively broad: LISP was used for modal logic, first-order predicate calculus theorem provers, education, "mechanical" anthropology (including text editing), spectroscopy, group theory, neurophysiology, automatic generation and testing of FORTRAN programs, and translation between different machine codes.

The appearance of the LISP 1.5 system from Stanford and its active use by physicists at the Rutherford Laboratory in Didcot inspired Dr. D.B. Russell of the Atlas Computer Laboratory in Chilton to build a similar system for the Atlas.

The Atlas Computer can be considered, in a way, as the first computer of the third generation, although it was not equipped with integrated circuits. This machine, developed at the University of Manchester, influenced all essential subsequent computers with its forward-looking architecture [LAV78].

The first version of Atlas LISP, LISPDEC66, was completed in mid-1966. It was largely compatible with LISP 1.5 on the IBM/7090: the user only had to remove the control cards for OVERLORD, and they could use their card deck in IBM format for execution on the Atlas.

Minor differences were: the functions ARRAY, TIME, TIME1, and ERRORSET, functions of the Car-Cdr family with more than three A or D between C and R, as well as all functions for character input and classification were not present. Car and Cdr of an atom could not be formed—errors resulted. Neither EQ nor EQUAL were meaningful for floating-point numbers: the comparison accuracy was exact to the last digit, and small rounding errors and input conversions led to different numbers (i.e., $1 \neq 1.0$). Although the first compiler was only "rudimentary," Russell had built a complete LAP based on the Atlas's assembler language, ABL [RUS66].

These capabilities were utilized by R. d'Inverno in his ALAM system [IN69], which became the most famous application program of the Atlas LISP system. The manual [IN68] simultaneously serves as a manual for the Atlas LISP. d'Inverno was unaware of Hearn's activities and found the symbolic computation programs available in mid-1966 for physicists in the field of gravitational theory too slow and therefore too weak. He programmed his relatively large system entirely in LAP—a unique case of using LISP.

In the course of 1967, Russell became acquainted with other LISP implementations. BBN-LISP on the SDS 940 seemed to have influenced him the most. Thus, he developed a second version of the Atlas LISP (LISP68), which,

however, seemed to be partially incompatible with the previous one. As Russell noted [RUS68], he could rely on the BBN program for the development of the compiler.

The LISP68 system is based on the binding scheme of LISP 1.6. However, how the compiler worked is not apparent because there was no declaration for free variables. The only possible explanation is that the compiler also treated every variable as free and secured access to the value cell. A whole series of functions were openly compiled—including not only the common FSUBR functions but also all members of the Car-Cdr family up to fourfold nesting, `CONS, RPLACA, RPLACD, EQ`, and ATOM, as well as the P-list functions PLIST (fetches the P-list) and ASCRIBE (sets the P-list) and some numerical functions (`ADD, SUB, INCR = ADD1, DECR = SUB1`). The system offered about 200 basic functions and overall represented a quite respectable product. The list space with a total of 100 blocks, each with 512 words (theoretically up to 4096 blocks), allowed for serious LISP activities.

In 1967, a group at Queen's University in Belfast also implemented a LISP system. This was done under the direction of F.V. McBride for the ICL 1905. This system was used for embedding pattern matching capabilities in LISP [MCB69].

For some years, LISP in the UK was mainly in use among physicists for formula manipulation purposes. Exceptions are the rare users of LISP 1.5 systems on IBM 7090 or 7094 systems (such as Luckham in Manchester, see [LUC67]). The list processing language POP2, developed in Edinburgh, gained more popularity among researchers in the field of Artificial Intelligence.

It was only in the mid-seventies that independent implementation activities were initiated. Interestingly, physicists started it once again: A.C. Norman had been working on small systems for the pdp-7 at the University of Cambridge since 1967. The experience with these systems and the unsatisfied desires with various other LISP systems led to the design of their own powerful system for the IBM 360-370.

The project started around 1974. The following basic decisions were made before program design: LISP 1.6 binding scheme, type characteristics in pointers to an object instead of in the object itself, compactifying garbage collection, comprehensive arithmetic capabilities, careful validity checks within each function. A methodological decision significantly influenced program creation: BCPL was to be used as the implementation language.

The report [FIT77] on the completed implementation describes an interesting system whose performance was not significantly below that of hand-coded STANFORD LISP/360. While the interpreter worked about 25% slower, compiled programs were about 20% faster. If arithmetic was used extensively, a factor of 2 (200%) could even be achieved.

Another independent activity in the UK was the implementation of InterLISP for the ICL-4 in Edinburgh around 1978. Under the leadership of G. Miller, some difficulties had to be overcome, such as a bug in garbage collection. When the system was ready, DEC-10 systems were delivered to Edinburgh, so no users of the ICL-4 InterLISP are known.

5.2 France

In France, as in the United Kingdom, work in the field of Artificial Intelligence began in the early sixties. Initially, the activities associated with LISP did not show promising results. J. Pitrat in Paris had access to an IBM 704 (at the Blaise Pascal Institute, see [FIN66]) and intended to work with LISP. A request for help sent to McCarthy did not reach its recipient or was not accepted; in any case, there was no response. Consequently, Pitrat turned to FLPL and worked with it.

In 1966, Kent brought his LISP system to the CDC 3600 at the same institute. It was primarily used for formula manipulation.

Since 1962, an IBM 7044 was operational in Grenoble, and LISP 1.5 was available from the beginning. Around 1963–64, T. Evans brought a newer version of LISP from Cambridge and conducted a programming course. The system was used by a small user group until the facility was dismantled in 1968. In 1963, an ALGOL compiler was constructed by a group of programmers and scientists led by J.C. Bousard. This base was then utilized by J. Cohen and his team for a LISP implementation [CO65]. From around early 1964, an ALGOL procedure system was built for embedding LISP in Algol60 [CO67b].

In the course of 1965, this system was used for both symbol manipulation work (with applications in formula manipulation and theorem proving) and experiments involving the use of external storage within a LISP environment. These experiments were quite encouraging, as a certain locality of the working sets of list pages could be determined [CO67a].

In his report on the utilization of the IBM 7044 system, Bousard reported that 5% of the total computing time went to users of COMIT, COBOL, or LISP [BOU66].

Between 1966 and 1968, it seems that LISP activity continued in Grenoble. One indicator is Ribbens' book [RIB69], who visited Grenoble during that time. Another piece of evidence is the subsequent history: Y. Siret took the first opportunity in 1968 to bring a STANFORD LISP/360 system from the USA to Grenoble when an IBM 360-67 was installed there. It was adapted to the local modification of the CP/CMS operating system. Siret left the institute in 1971, and with his departure, all activities were neglected and ceased.

In 1973, this system was taken care of by A. Lux, who organized the chaotic program structures. He completed his doctorate in 1975 with a thesis on the LISP implementation using a virtual LISP base machine [LUX75a]. He applied these ideas to construct the new LISP system in Grenoble, which ran on IRIS80 after the old IBM/360-67 was replaced ([LUX78], also see [LUX75b]).

Since around 1970, P. Greussay's group at the University of Paris 8 (Vincennes) has been active. They started their own implementation in 1971: a LISP system for the CAE-510, a domestic 18-bit/8K computer. This system had many characteristics of LISP 1.6 [GREU72]. [See https://www.artinfo-musinfo.org/en/issues/vlisp/Lisp510.html .]

The experiences with the small computer led to ideas for improving the interpretation (no CONS operation during interpretation due to system require-

ments, etc.). Soon after, a desktop computer CAB-500 with 16K drum memory and 32-bit words was chosen as the basis for a renewed LISP implementation.

The improvements made so far led to the view that they had achieved an independent development: thus, the improved system on the minicomputer CAE-510 was named VLISP (Vincennes LISP). In addition to the usual methods for space saving and acceleration, Greussay extended the concept of the pattern for the interpretation of LISP to the so-called filter [GREU76].

In 1974, the group, which now also included H. Wertz, J. Chailloux, D. Goossens, and others, had the opportunity to use a Telemechanique T1600 at the Institute de l'Environment for LISP implementation. This system was completed around 1975 [GREU75].

Greussay formulated the basic ideas of VLISP in 1976, just as they began to implement VLISP on a pdp-10 [GREU76]. By mid-1976, this system was completed [CHA76]. VLISP-10, as it was called, was a quite extensive system. It included about 200 basic functions and a range of useful tools, such as the testing system PHENARETE [WER76,77,78], which was comparable to DWIM and other similar products.

The data types included atom symbols (up to 18 characters), character strings, integers, and lists. The function types were SUBR, NSUBR (similar to LSUBR in LISP 1.6), FSUBR, EXPR, FEXPR, and two macro types: MACIN and MACOUT. By noting an atom in the variable list's position, a LEXPR processing could be initiated, similar to MacLISP. However, VLISP did not offer the auxiliary functions SETARG and ARG; instead, it assigned the entire argument list to the atom.

Among the functions, the following developments stood out: EPROGN took a list of expressions as an argument, which were sequentially evaluated. As with PROGN, the last expression determined the overall value. A function IF with n arguments offered simple COND possibilities: if the first argument does not yield NIL during evaluation, the evaluation of the second argument provides the overall value. In other cases (first argument is NIL), arguments 3 to n were sequentially evaluated, and the value of the last one was used as the overall value.

In addition, there were functions for selection (SELECT and SELECTQ) and cyclic evaluation (WHILE, UNTIL, and REPEAT). Two functions served as global exits: ESCAPE and LESCAPE. While ESCAPE introduced an atom as the first argument as exit functions, whose call terminated the entire ESCAPE expression, regardless of how deeply nested this call was, LESCAPE served to exit the last LAMBDA expression.

VLISP included a range of functions and a generous assortment of basic functions for list processing. A stack accessible only to the user could be managed with two access functions PUSH and POP. The arithmetic functions had partially unusual names, and the spectrum corresponded to LISP 1.6.

Regarding input and output, the LISP 1.6 standard was once again provided. New were the input and output macros: for example, a input macro could be defined using DMI:

(DMI IF (B THEN . ELSE) (LIST 'COND (LIST B THEN) (CONS T

ELSE)))

Now, whenever IF was read, the subsequent element was bound to B, the next element to THEN, and the entire remaining list to ELSE. Then the function body was evaluated, and the result was considered the read result. For example, given the expression:

(IF(GT A B) (SETQ X B) SETQ X A) (INCR Y))

It results in:

(COND ((GT A B) (SETQ X B)) (T (SETQ X A) (INCR Y)))

Similarly, but concerning output, the work with output macros proceeded: by

(DMO QUOTE X (PRIN1 (QUOTE ') X))

It could be ensured that during output, instead of the atom QUOTE, the apostrophe would be used.

String manipulation was based on the usual basic functions for type checking, transforming strings into lists, comparison, concatenation, duplication, reversal of strings, and searching for substrings, replacing them, and determining their position within a comprehensive string.

Remarkably, VLISP had an output option in Braille.

Further improvements were naturally made to the finished VLISP. For example, in May 1977, a description of the new capabilities was presented [CHA77], which now also included floating-point numbers, processing macros (defined with DM), and arrays (defined with DA). In addition, the auxiliary programs (editor, Phenarete, error correction) and the compiler (with LAP) were improved or even used for the first time.

With this system, they now had an advanced working tool. The successes in the implementation encouraged further steps (implementation of VLISP for the Intel 8080 microprocessor) and the transfer of the pdp-10 system to a DEC-20 at the Open University in England.

The research group under Greussay was one of the most active in Europe at that time. This is also evident from the publication of the LISP Bulletin by Greussay in the summer of 1978.

In addition to Paris, Toulouse must be mentioned, where under J. L. Durieux, a LISP implementation TLISP was developed [DU].

Theoretical work on LISP was also carried out in Paris-Rocquencourt [DER77a, DER77b].

5.3 Netherlands

In the Netherlands, the history of LISP is shaped by two individuals, W.L. van der Poel and G. van der Mey.

Although van der Poel became aware of the article [MCC60c], he initially did not seem to find LISP too interesting. In any case, around 1962 or 1963, he commissioned van der Mey to write an interpreter for the IPL-V programming

language. While van der Mey was working on the program, van der Poel had the opportunity to spend an extended period in the fall semester of 1964 at Stanford University. There, he became acquainted with LISP through McCarthy. The study of use cases and the simplicity of the language fascinated him so much that upon his return, he went to van der Mey with the enthusiastic news that he had discovered a much more beautiful programming language.

"He did not allow himself any time to test my IPL program but instructed me to write an interpreter for LISP 1.5 for use on the pdp-8 machine" [MEY78]. Here, van der Mey does not remember entirely correctly: The first system, according to van der Poel, was intended for ZEBRA.

This program was completed in 1965 and was initially tested in the laboratory of the Dutch PTT. It was a relatively modest program that only provided a 1K list memory. Many of the usual basic functions were omitted, but members of the CAR-CDR family up to 11-fold nesting were present. Shortly thereafter, the implementation for the pdp-8 began.

The user worked in both systems with EVALQUOTE, so pairs had to be provided: functions and lists of arguments. These inputs were accepted as punched tapes or directly from the teleprinter. The implementation technique is interesting here because the stack memory was located within the list memory; "In fact, the stack itself is organized as a list in the normal list memory..." [MEY67]. The complex memory space problems were attempted to be solved by having all standard atoms simultaneously represent functions. Thus, the function NIL has as many arguments as desired and always returns NIL, the function THAT takes one argument and returns its value, the function APVAL is used to search for spaces on the input punched tape, and so on. EXPR was even used as a multipurpose function that could jump to a specific location in the main memory to execute a piece of code. Here, the user could prepare their own sequences of commands but also had certain standard variants. These included: switching states to accept or output numbers in octal or decimal, masking or logical operations on numbers, printing individual characters, and so on.

For system functions, instead of the P-list, the value of their atom pointer was used to represent the relevant property (start of the code).

A special innovation, based on the definitions of functions NIL and T, was possible: Conditional expressions could be written without having to use COND as the function name. An expression sequence like:

```
((NULL X) (RETURN Y))
((NULL Y) (RETURN X))
(T (RETURN (CONS X Y))
```

was allowed and processed as follows: First, (NULL X) is evaluated to determine the function for the first expression. Suppose it results in NIL. The function NIL, however, returns NIL without further examining its arguments. This leads to the second expression. Similar things happen here. Finally, the function T evaluates its argument and triggers the RETURN in this way.

Van der Mey's pdp-8 LISP was included in the library of the user community of the Digital Equipment company, DECUS, around 1968 [MEY68] and was

considered a model for LISP implementation on small computers for a long time [NAK69].

In the course of 1966, the PTT laboratory received an Electrologica-X8. Van der Poel suggested to van der Mey that they should also build a LISP system for this significantly larger machine.

Van der Mey did not want the LISP implementation to become routine, and he thought about improvements that extended to the language level. Certain limitations in the input and output of cyclic lists or lists with physically identical sublists seemed to require modification to him, especially. Indeed, structures could easily be created using RPLACD or RPLACA that were not notatable in the S-language.

Since the EL-X8 had enough free bits for marking purposes, van der Mey developed a printing program in which the structure to be output is first searched for identical branches, then the lists and marked sublists are output, and finally, the structure is restored to its original state.

Thus, the list whose elements are (A B), A, and B, is output as:

((A B) A B)

However, if we perform an:

(RPLACD Liste (CAR Liste)),

then in all conventional systems, the same character string is printed. In the EL-X8-LISP, however, it is printed as:

((3000: A B) :3000)

[KAL75][S.105].

[KAL75][p.105].

Interestingly, van der Mey considered the same marking for garbage collection as an improvement over the marking in pdp-8 LISP.

The EL-X8 system was completed around 1967 [POE67]. It was delivered to Kiel in 1968 [KAL75].

The available machine park changed when van der Poel was appointed a professor at the Technical University in Delft in 1968. There, in addition to the pdp-8, they also had a pdp-9. It was clear that van der Poel, who included list processing in his lecture program [POE72], also wanted an implementation for this system.

Initially, they (possibly C.F. Knoet) simply rewrote the LISP system from the pdp-8 [MEY70b]. But van der Mey had new ideas: he wanted to use the three bits freely available compared to the pdp-8 to realize a compact list representation. It is not unlikely that this project was contributed to not only by the shortage of memory on the computer but also by W.J. Hansen's publication from 1969 [HAN69].

It was assumed that in most cases, only real lists would be used. When these were stored physically close together, the CDRs always pointed to the

next cell, i.e., to the next machine word. Only the last CDR contained NIL. These lists were represented so that only the CARs were allocated words, with markings indicating that it is a genuine list element. In addition, the last element received the note that the associated CDR is NIL. In van der Mey's terminology, therefore, every normal list could be represented by open cells and an end cell [MEY71].

However, problems arose with the CONS and RPLACD operations. These were solved by introducing bridge cells that couple the physically scattered parts of the list. To not lose the advantages of linear arrangement due to unpredictable processing operations, a linearizing garbage collector was programmed. This first marks the active cells, then pushes the entire list memory together, and copies the lists to free space to linearize them.

This system, called HLISP [MEY70a], was functional around 1974. Van der Mey discovered that the regular linear arrangement of lists offered possibilities for accelerating the processing of recursive processes [MEY71]. He published his ideas in the same year.

Since that time, van der Mey has not been directly connected with the LISP implementation.

Under van der Poel, work continued in Delft. The pdp-8 LISP was rewritten and improved several times. Opportunities for this mainly arose from improved main memory conditions: A version for 8K was developed by A. Kooiman and M.R. Vlendr'e, another, significantly revised version (Cars and CDRs are not directly adjacent but in two separate fields) for 16K by A.J. Mesman and H.W. van der Poel, the son of W.L. van der Poel [MEY74].

In 1974-1975, van der Poel supervised the doctoral thesis of K. Müller on concurrent garbage collection [MUE76]. This work was completed in August 1975 and was independent of Steele's work [STE75] and a group around Dijkstra [DI75] (see below) that occurred around the same time. Müller's efforts do not specifically relate to LISP but are applicable to arbitrary data structures, although they allow comparatively less parallel work than the other authors.

In 1977, M. den Hoed created a LISP implementation for the pdp-11 [HOE77]. He used the system programming language HARVALIAS for this [EY75]. This system was largely based on the pdp-8 systems, allowing circular structures when reading and printing, and used similar function definitions for NIL and T.

The work of the Dijkstra group, in which the Dutch A.J. Martin, C.S. Scholten, and E.F.M. Steffens participated, on the development of a highly parallel garbage collector working alongside the list processing system, should be mentioned at the end. Although not directly intended for a LISP implementation, it will certainly influence future work.

5.4 Norway

In Norway, interest in LISP began around 1965. Presumably, there were connections from Oslo to K. Korsvold, who was involved in Stanford formula manipulation with LISP. L. Monrad-Krohn commissioned Jan Kent to write a master's thesis on LISP implementation and to implement it on the available CDC 3600.

Kent had access to a listing of the 7090 LISP (which version is no longer clear), and he largely followed the model in his programming. By December 1966, the system was sufficiently stable, and Kent completed his Master of Science in Mathematics examinations. The written work, practically representing a description of the LISP system for the CDC 3600 [KEN66], was published as an Internal Report of the Norwegian Research Council at the Ministry of Defense.

After a short time, i.e., still in 1966, Kent's system was already distributed by the user organization of the CDC computers, CO-OP. Kent himself had conducted his work on the machine in Kjeller, Norway. He brought it to the Blaise Pascal Institute in Paris, the Technical University in Stockholm, and Uppsala University.

In early 1967, Kent traveled to the USA and Canada. He took the CDC 3600-LISP system with him and installed it himself at Michigan State University and the University of Waterloo in Canada. The CDC 36000 system must have been widely used. Over 100 copies of the system description were printed and sent to users.

In Waterloo, Kent gave a lecture on list processing and began planning a LISP implementation for the IBM/360-40 there. His ideas were supposed to be published in the then-unpublished book by Berkeley and Edwards, "The nature, use, and implementation of the computer programming language LISP" [KEN67, BE67].

Together with J.F. Bolce and R.H. Cooper from Waterloo, Kent then began the implementation on the IBM 360. Once the system was largely functional, Kent was invited to Stanford to complete the implementation on their machine. Together with Berns, the system was completed during the year and later became known as STANFORD LISP/360 (see p.?). In Waterloo, Bolce and Cooper also completed the interpreter for the IBM 360 in the same period [BOL68]. Both systems can be considered identical. However, as noted elsewhere, Berns continued to improve the system for a longer period in Stanford.

Around 1968–69, Kent returned to Oslo. There, he became the client for a master's thesis, this time with the student E. Holbaek-Hansen. The task was to design and program a LISP system for the CDC 3300. This work was completed around the end of 1970 and published by the University of Oslo [HOL71a]. In early 1971, the LISP 3300 could be obtained abroad – this is confirmed for the University of Erlangen-Nuremberg.

Although this LISP system did not reach a very large user base, work on it continued in Oslo. Graduate students had their share in this development, but Holbaek-Hansen seems to have contributed the most. The user manuals from 1972 [GOE72] and 1973 reflect this development [HOL71b].

Around 1973, the system was expanded with a LAP and a compiler (Minicomp) [HOL74a]. Finally, by the end of 1974, an overall revised version was available [HOL74b]. It included two compilers: the Minicomp, a small, restricted but extremely efficient compiler, and a larger, more extensive one, about which little can be found in [HOL74a]. Both compilers were programmed in LISP – just like the LAP translator.

Another LISP implementation under Kent's supervision began in late 1973:

A. Eide and H.K. Langva created a LISP 1.5 system for the CYBER-74. Their master's theses were published in 1974 [EID74, LAN74].

In 1976, Kent worked on a LISP implementation for microcomputers.

His system for the CDC 3600 is said to have served as a model for various implementations for the CDC 3400.

5.5 Sweden

During the year 1967, Uppsala University acquired the LISP 1.5 system for the CDC 3600. They began using it for simple problems and for teaching purposes. From 1968, individual students attempted to enhance the system. Work was done on the compiler, LAP [MAK69], and the runtime system [NOD68]. By mid-1969, so many changes had been made that a new manual had to be issued [U70].

After gaining some knowledge of LISP, they started to scrutinize the existing system. Sandewall spent some time in the USA, and Urmi was at BBN in Cambridge in 1970. They planned a LISP system for a Siemens 305, but the computer did not arrive. To work on other computers in Sweden through RAX remote processing, they came up with the idea of creating a LISP system in FORTRAN. They began this in January 1970, and by September, M. Nordström and Sandewall had written a reasonably acceptable system in FORTRAN IV for IBM machines [NOD70].

This system was shortly expanded by an editor and the TRACE function, both from D. Breslaw. Although LISP F1 was intended for interactive work, it could also run in batch mode. It roughly corresponded to LISP 1.5 but already included some extensions from the BBN-LISP repertoire, such as no-spread LAMBDA expressions, BREAK functions, and user-controllable error handling.

The memory requirements were based on the 36K bytes needed for the program and its workspace, to which dynamic memory space was added (approximately 45K and more). Names were limited to eight characters, and floating-point numbers were not available. The user could choose between the EVAL mode and the EVALQUOTE mode. The system offered 75 functions, among which the definition functions DE and DF appeared for the first time here. They were later incorporated into many other LISP systems.

When Urmi returned from BBN, he began implementing a BBN-LISP-like system for the process computer Siemens 305. This system did not have a compiler but allowed the study of interactive work.

In January 1970, Sandewall, along with eight interested researchers (including Nordström, Urmi, O. Willen, and A. Haraldson), founded the Datalogilaboratoriet. The main focus was on issues related to question-answer systems, formula manipulation, and sociological models [DAT70]. Haraldson and Breslaw, alongside Urmi, worked on the Siemens system, which was conceived similarly to BBN-LISP as page-oriented. PILS (Paged Interactive LISP System) was intended to operate in a virtual 21-bit address space and provide access for at least four users simultaneously.

Arrangements were made with BBN for the delivery of many system programs, allowing them to adopt entire subsystems like the editor, DWIM, and Programmer's Assistant. Although planned, according to Urmi, the compiler was not completed. The entire system for Siemens 305 was considered complete in 1972 [DAT75].

The unexpected termination of the PILS project was due to the planned purchase of an IBM 370 by Uppsala University, which had been discussed since the summer of 1971. At the end of 1971, the group decided to build a powerful LISP implementation on this system, applying their previous experiences. That is, constructing a BBN-LISP-compatible system based on a software paging mechanism.

The project was planned in collaboration with the Uppsala University Computing Center (UDAC) and carefully prepared. The facility, an IBM System/370-155, could be used for implementation work since the end of 1971. However, the group, consisting of Urmi, M. Andersson, and B. Jansson, needed until around 1975 to overcome all difficulties. Typically, an effort of 3 to 5 man-years is cited.

The distribution of the system, which was named InterLISP 360/370 since 1974 [U75], was taken over by UDAC. It was resold for 40,000 Swedish Kronor, with a certain percentage having to be paid to BBN. Deliveries are known to the University of Zürich, Hannover Medical School, LAAS in Toulouse, and the Weizmann Institute in Israel.

In 1974, Siemens in Munich proposed transferring InterLISP/360-370 to Siemens 4004 systems. Since the machines had the same internal structure and identical assembly language, it was believed that the transfer could be completed quickly. However, certain tricky differences in the operating systems complicated the process.

Although, according to Urmi, only a few weeks were needed to resolve these issues, Siemens only took over responsibility in the fall of 1977.

In the course of 1977, the InterLISP system was also delivered to Japan. Initially (in October 1977), it ran on an IBM System/370-158 from Fujitsu Ltd. and another IBM System/370-138; in February, it could be taken over on Fujitsu M160 or M190 systems.

The FORTRAN system F1 was also maintained. Significant increases in efficiency were achieved initially [DAT74], and in the summer of 1973, Nordström coded a completely revised system. This system was now three times faster, included some of the usual LISP functions from other systems for input and output, and distributed others for input and output. By 1976 (2.1), the number of user installations exceeded 96 ([DAT75], p.20).

In a renewed attempt, a subset of InterLISP was coded in FORTRAN in the summer of 1976 (LISP F3). Once again, Nordström was the author. LISP F3 included about 100 basic functions and a series of program packages from InterLISP, such as EDIT, MAKEILE, BREAK, ADVISE, and TRACE, as well as packages of other fundamental functions.

Since 1975, the researchers have separated: a group under Sandewall moved from Uppsala to Linköping. The areas of work remain connected with LISP. Sandewall turned to methodological aspects of LISP programming [SAN75b,

SAN76, SAN78]. Urmi completed his doctoral thesis [U78], as did Haraldson [HAR77a]. InterLISP/360-370 was passed on to the Weizmann Institute in Israel around 1978 for further maintenance.

With the import of American machines, other LISP systems were, of course, also imported to Sweden. Documented are STANFORD LISP 7360, STANFORD LISP 1.6, and InterLISP on DEC-20.

5.6 Federal Republic of Germany

Around 1962, the first IBM7090 systems were delivered to the Federal Republic of Germany. For the second machine that went to the Gesellschaft für Mathematik und Datenverarbeitung (GMD) in Bonn, C.A. Petri, with the help of his private connections to MIT, obtained a LISP 1.5 system. This was tested in early January 1964. However, Petri argues that this, and the versions created by students based on publications at that time (using FAP macros), cannot be considered as a complete adoption into the Federal Republic of Germany [PEI78].

In early 1964, Hearn sent a magnetic tape with LISP 1.5 and his initial formula manipulation system to physicist Weicksel at the Institute for Theoretical Physics at the University of Mainz. It is not known whether it was actively used. Later that same year, Hearn himself arrived with a new version. He used the IBM7090 in Darmstadt, and it is highly likely that work continued after his stay.

At the Institute for Instrumental Mathematics at the University of Bonn, LISP 1.5 was installed for general use on the IBM7090 in 1965; it had been acquired through SHARE or the European Program Library as early as possible. In the winter semester of 1965/66, the "Data Processing Seminar" was mainly dedicated to the introduction to LISP and practical exercises. In the following semesters, LISP 1.5 became a regular part of lectures.

The first independent LISP implementation (or adaptation) in Germany was carried out after 1967 by H.D. Baumann in Geesthacht for a CDC3400. He followed the ideas of Kent and his implementation for the CDC3600.

Around 1967, H. Hünke in Bonn started a LISP implementation for the IBM/360. Based on Genrich's work on METEOR, he attempted to integrate this system from the beginning. The system IDOL, an interactive display-oriented list processing system, ran in the B.I.T.S. operating system, self-constructed by GMD [HUE68]. Although an interesting system, it seems not to have found lively use.

Regarding the use of LISP 1.5 in Darmstadt, it is known that physicists primarily worked on REDUCE Feynman diagrams or problems in quantum mechanics. Additionally, text processing, mathematical text tasks, and games were performed.

By 1967, access to IBM/360 machines was available, and the STANFORD LISP/360 was acquired. REDUCE, distributed by Hearn since 1968, was also procured. However, LISP was not only used; there were considerations about testing and managing LISP programs [BA68].

In 1968, the University of Kiel adopted the LISP system from the Netherlands for the EL-X8 [KAL75]. Substantial work was done on this system in Kiel: it was adapted to the specific conditions and significantly extended. Interestingly, Jürgensen mentions experimenting with German's pdp-1-LISP before commissioning the VANDER MEY system. Jürgensen wrote a pdp-1 simulator and worked through the original program [DEU64].

All these efforts were under the direction of H. Jürgensen, who not only supervised the work of F. Simon and H. Kalhoff but also wrote the first user documentation, among other things [JUE70]. Simon established an "automatic secondary storage organization for storing S-expressions and compiled functions" [SIMO70]. Kalhoff is the author of the compiler for the X8-LISP [KAL72].

Around 1970, H. Melenk started implementing LISP 1.5 for the TR4 in Marburg. This system is heavily inspired by STANFORD LISP/360 or VAN DER MEY's pdp-8-LISP, so it can be assumed that this implementation existed around 1970. A manual was prepared around 1972 [MEL72]. MR-LISP, as Melenk named his system [LUE73], was supposed to handle relatively large programs, even though the TR4 only provided 18K. Therefore, a special compiler was developed, generating a linear sequence of pseudo-instructions and outputting them to an external memory. A page memory managed by software was developed for these translated functions. A special interpreter for pseudo-instructions executes the loaded functions [LUE73].

Around 1972, Melenk moved to the GRZ for Science in Berlin (West). There, he and R. Roitzsch began an implementation for the TR440. They tried to develop a LISP system that was particularly suitable for interactive work on consoles and display devices.

Special attention was paid to error messages and testing aids [MEL73]. This system also had a compiler in the same way as its predecessor on the TR4.

Since the system continued to be maintained, the improvements were summarized in a new edition of the manual in 1975 [ROI75]. Specifically, it included expanded arithmetic, new input and output functions, and improved system components (faster interpreter and compiler, demand-paging memory management). This spectrum suggests the main application area of the system: formula manipulation. In G. Görz's list of "LISP systems in the FRG" [GOE76], this system is referred to as B&LISP.

This list also shows the large number of foreign systems that were used in the Federal Republic of Germany in 1976. In addition to those already mentioned: STANFORD LISP/360 (mainly in conjunction with the formula manipulation system REDUCE), UNIVAC LISP (p.234), STANFORD LISP 1.6 (p.222), UTLISP (p.232), CDC3300-LISP (p.), LISP F1 (p.245), LISP F2 (p.246), and finally, LISP 1.6 for the B-6700 (p.266).

Since 1973, a sharply increased interest in LISP is noticeable. Research groups for Artificial Intelligence emerged in Kiel, Hamburg, Bonn, Mannheim, Karlsruhe, Stuttgart, Nürnberg-Erlangen, and Munich. While Kiel, Hamburg, Mannheim, and Karlsruhe contented themselves with existing systems or tried to obtain better systems, new systems were developed in the other cities. As part of diploma theses, M. Leppert in Munich [LEP74], R. Woitok in Erlangen

[WOI75] - both on the TR440 - developed systems roughly in the size of LISP 1.5. At the same time in Regensburg, LISP 1.5 was developed for the Siemens 4404/45 (BS 1000).

In Stuttgart, a group led by H.J. Laubsch began implementing MacLISP on the TR440. This system should have been operational in 1975; a manual [LAU76] was produced in 1976. Obviously designed as a subset of MIT's MacLISP [MOO74], MacLISP already had 200 basic functions in March 1976. This included basic functions for graphical output on screens as well as (programmed in LISP) program packages for editing, tracing, pretty-printing, and other testing aids.

During 1976, one-dimensional arrays (including OBARRAY arrays for exchanging READ environments) and some new functions were added. The most interesting changes are the three functions FUNCTION, CLOSURE, and EVALFRAME.

FUNCTION provides a FUNARG expression that is only useful for deeper nesting. However, the relevant functional argument cannot be passed as a value outward. On the other hand, CLOSURE generates a real FUNARG expression. It corresponds to the solution proposed by Sandewall for the FUNARG problem [SAN70], where a list of free variables needs to be specified, and the defining environment should be passed along. Assignments are possible for these variables, allowing a kind of coroutine concept to be realized.

EVALFRAME makes all dynamically higher forms of EVAL and their environments accessible. This allows the implementation of complex control structures. However, the normal interpreter does not build these environments. A special mode must be activated for this purpose. EVALFRAME returns a list of three elements as values, where the first is a stack pointer, the second is the associated form, and the third is a pointer in the environment. These values can be referred to, for example, with FRETURN, which evaluates its second argument in the environment specified by the stack pointer (first argument) and then jumps back there [LAU77].

Around 1978, the development of a LISP system for the Interdata M85 or 7/32 in Kaiserslautern is reported.

However, the most significant development in the Federal Republic of Germany around 1975-77 was the adoption of InterLISP by Siemens. This process was significantly triggered by the Institute for German Language in Mannheim. In connection with the "Linguistic Data Processing" project, efforts were made to obtain various LISP systems. MTS-LISP from Ann Arbor was procured for this purpose (p.231), and in February 1974, LISP F2 from Uppsala was acquired. This was implemented at the Institute for Information Processing in Technology and Biology (IITB) in Karlsruhe. From this system, the Bonn branch developed the version LISP FINT, which stands somewhat between the F2 and F3 versions (p.246) and was intended to represent a kind of InterLISP without testing aids.

Soon the idea arose to adopt the entire system. Urmi visited Mannheim and explained the details of the system. Since the system had a price of 40,000 Kronen, the idea came up to interest Siemens in the purchase and adoption for the Siemens 4004.151. This succeeded, and together with partners from

Uppsala, they worked on transferring InterLISP to the BS2000 operating system. After overcoming initial difficulties, the first version could be handed over to users at the end of 1974/beginning of 1975.

Involved in this work were T. Krumnack, T. Christaller, C. Schwarz, D. Kolb, as well as Scott, Sievert, Vokand, and Ritzer. In early 1975, B. Epp wrote the first programming manual [EP77]. Siemens-InterLISP [GU78a], as the system has been called since then, has been revised until 1978 and is now available in version 3 [GU78b]. It corresponds to the state of InterLISP for the pdp-10 from 1972.

Similar to InterLISP 360/370, the system is divided into two essential components: the base system includes all LISP primitive functions and all operating system-dependent routines for memory and resource management. This base system is implemented in assembler language using the SPL runtime system. The development system defines most of the Siemens-InterLISP language repertoire. It includes function packages like a compiler, editor, BREAK package, ADVISE and TRACE, etc., and is written in LISP. The base system accounts for less than 10

The plans in 1978 concerned both parts of the base system and the expansion of the development system. The base system should become faster through the integration of word cells. The garbage collector should be developed into a parallel real-time garbage collector. The use of direct access files and index-sequential files was intended as further improvement.

The expansion system should incorporate DWIM. The compiler should be improved; curiously, the generation of "more optimal code" was expected ([GU78b], p.17). There were plans to adopt the block compiler. Finally, they hoped to use LAP for simulating machines ("future architecture of novel hardware"). For this purpose, a LAP interpreter should be developed.

5.7 German Democratic Republic (DDR)

In the German Democratic Republic (DDR), research in the field of Artificial Intelligence began around 1969, shortly after the establishment of the Robotron combine. The small working group, led by E. Lehmann, was placed within the overall context of "Automation of Programming." It became apparent quite soon that the foundational system for the desired practical work could only be LISP.

In 1970, P. String initiated the design of an interpreter. This task was not easy because the entire information about LISP was contained in the well-known book by Berkeley and Bobrow [BB64]. Other information seemed unavailable.

When a computer became available in mid-1970, H. Stoyan, independently of String's approach, began evaluating the article by Deutsch on pdp-1-lisp [DEU65]. The first interpreter was practically an adaptation of the system from Deutsch to the IBM-compatible ESER facilities. Despite extremely limited computing time (2 hours per week), a very restricted system was operational in 1970; LISP 1.5 (without arrays and floating-point arithmetic) was achieved around mid-1971.

At that time, the author was anything but a fan of LISP and was less willing to accept extensions. However, Lehmann and other colleagues needed a more powerful system for their work. String then started implementing the Q-32 Compiler [SAU64c] for ESER computers, while the author, along with collaborators I. Köhler and S. Kühnel, gradually brought the LISP system to the LISP 1.6 level. When the first compiler tests looked promising at the end of 1971, the author planned the Run-Time System for the compiler and proposed a hand compilation of the compiler written in LISP up to that point. This was executed, and the resulting program consistently demonstrated remarkable speed.

By early 1972, the LISP 1.6 system was completed. At that time, it did not include arrays, only integer arithmetic, and did not allow for changing input and output streams. However, it featured direct access, which included external addresses of relocated properties in P-lists, automatically loaded during GET operations. The Compiler was connected to this file, storing the compiled programs in machine format (along with a list of literals in external LISP format). During execution, the functions were automatically loaded from the disk file and managed according to user-defined priority principles.

In early 1973, work in the field of Artificial Intelligence was halted. Only a small group continued to focus on question-answer systems.

In 1974, the author moved to the Dresden University of Technology. Due to a series of coincidences, a situation arose where further development or resumption of LISP work seemed opportune. The process began with a complete restructuring and revision of the system, making future changes easier, and rendering the system more manageable and modular. A speed test (see page?) yielded a time of 1460 ms for a list of 20 elements and 62400 ms for a list of 40 elements.

A new acceleration approach was initiated in early 1975. Until then, the value cells (SPECIAL cells) were ordinary properties associated with the corresponding SPECIAL indicator. This was changed so that each atom in the atom head carried the value. Simultaneously, type descriptions were moved from the data cells to the pointer to the data cells. The system now achieved a time of 1000 ms for the same test with a list of 20 elements and 3620 ms for a list of 40 elements.

Together with students Steinborn and Neuhaus, parameter transmission was completely reworked by summer 1975. Until then, a list of evaluated arguments had been formed, with each subsequent one being inserted at the end. Instead, the evaluated arguments were now arranged in the interpreter regime as well as for compiled functions in the stack memory. Simultaneously, functions like LIST and TIMES were renamed from FSUBR functions to LSUBR functions. For error monitoring purposes, the arguments were counted, and in case of a mismatch between the delivered and required argument numbers, errors were reported (in the case of LSUBR functions, only the current arguments were counted for uniformity in the central routine EVLIS). The measurements now resulted in 720 ms for the shorter list (20 elements) and 2620 ms for the longer list (40 elements).

Around the summer of 1975, the author received the assembler listing of STANFORD LISP for the first time. Comparisons of code segments revealed certain advantages of STANFORD LISP/360 and other advantages of DOS/ES LISP 1.6. The differences were carefully evaluated, and the affected parts of the system were rewritten. However, these were not central processing programs, but, for example, garbage collection (the idea of not checking the end of the free memory list in CONS but generating a program error was already taken from the documentation of Waterloo LISP/360 [BOL68] in early 1975).

In early 1976, the author learned about the efficiency comparisons by T. Kurokawa [KUR76c]. To achieve an "honorable" comparison value (all other measurements were made only then), all brakes were carefully sought and removed once again. The system now performed no CONS operations during interpretation, calculated the maximum number of required stack memory slots at the point where it was clear that stacking was necessary, and stacked from there without checking, etc. Through meticulous command counting and replacement, the system eventually achieved 580 ms for the 20-element list and 2120 ms for the 40-element list. These values proved to be unbeatable.

By introducing measuring instruments according to InterLISP-BREAK-DOWN, which were intended to influence processing as little as possible, a negligible runtime deterioration of 680 and 2540 ms had to be accepted in early 1978. The compiler produces programs that require 98 and 340 ms for the same examples.

In early 1977, experiments for software paging of the list memory were conducted jointly with S. Kanew [KAN78] regarding his model of the software pager for DOS/ES. The system, carefully tailored to minimal use of list memory, required 2520 ms for the same examples as above. or 9580 ms[18].

Until 1979, DOS/ES LISP 1.6 had grown significantly. It included strings and arrays of any number of dimensions (similarly, the number of arguments for functions was unlimited)[19], over 400 functions, and an overall internationally comparable design [STO78].

5.8 Poland

At the Conference on Symbol Manipulation Languages and Techniques from September 5-7, 1966, in Pisa, S. Waligorski introduced the string processing language LOGOL. However, he was confronted with such a multitude of implementations and applications of LISP that he became convinced of its quality and engaged in discussions with implementers. As a result, he returned with the intention to implement LISP.

In early 1967, Cohen's article on embedding LISP in ALGOL [CO67a] was published, and Waligorski decided to pursue this path. He had the GIER-

[18]For comparison: LISP 1.9 achieves times of 263 ms and 946 ms on TOSBAC-5600; UT-LISP achieves 389 ms on CDC 6600 for the shorter list; STANFORD LISP 360 on IBM System/360-67 achieves 960 ms and 3580 ms; STANFORD LISP 1.6 on DEC-20 achieves 568 ms and 2079 ms; LISP 3300 on CDC3300 achieves 4751 ms and 20029 ms; UT-LISP on CYBER-172 achieves 894 ms and 5728 ms.

[19]LISP 1.5 limited to 20 arguments, MacLISP limited to 5 arguments, STANFORD LISP/360 limited to 22 arguments.

ALGOL, a famous efficient system [NAU67]. Collaborating with students H. Zelman, J. Zamorski, and M. Pluciak until around 1968, a LISP system was embedded into the ALGOL4 version. It appears that Zelman performed the actual programming and testing work [Z69]. This system was used for teaching at the University of Warsaw [WALI70].

Similar to Cohen's system, the fundamental functions were realized as ALGOL procedures. Lists were represented in an array, and indices were used as pointers. The list memory ranged between 2 and 4K.

Inspired by Waligorski's enthusiasm and encouraged by Z. Pawlak, J. Loska and his assistant B. Zochowska started implementing LISP 1.5 on the Odra 1204 in 1968. This computer achieved approximately 200,000 operations per second and had 64K of main memory. The system, usable by 1970, closely resembled the original LISP 1.5 [MCC62c]. Additionally, access via displays was provided.

The system was intended for research in circuit design or theoretical work on data structures. However, it seems there was not a large user community.

Nevertheless, in 1974, Loska and Zochowska initiated a new project: this time, they aimed to implement STANFORD LISP 1.6 on the Odra 1304 or 1305.

By 1975, this system encompassed nearly everything found in STANFORD LISP1.6. Exceptions were the EXPLODE, BOOLE, and DM functions (i.e., no macros), as well as functions for swapping input and output channels. Another limitation was in arithmetic, as only integers were allowed.

As work on the system continued, it was completed around 1976.

Around 1974, Bartnik developed a LISP system for formula manipulation on the KAR65 at the University of Warsaw (Physics Section).

A larger group dedicated to LISP implementation and application was founded in Poznań in 1973. They started implementing LISP for the K-202 minicomputer. Although the system was named LISP 1.5 [BER73], it included functions like READCH, RATOM, allowed the switching of input channels, etc. A compiler was also written. In 1974, M. Berlog, B. Begier, J. Kniat, and M. Stroinski turned their attention to the Odra 1204, for which a roughly compatible system was created [BER74].

5.9 USSR

Although LISP had been acknowledged in the USSR before 1963 [FEI61], its actual adoption began after 1967. One reason for this late start can be attributed to the lack of popularity of computer applications for non-numeric purposes. However, Turchin [TUR68, 77] had developed an excellent system tailored to the BESM-6 for non-numeric information processing, named REFAL. For a long time, there were claims in the USSR that this system was much better and more cost-effective than LISP.

The introduction of LISP to the USSR occurred when S.S. Lawrow became acquainted with Berkeley in 1967. Berkeley was an outspoken LISP enthusiast. He inspired Lawrow, and during Berkeley's visit to Moscow (presumably for an

international conference) in 1967, they jointly programmed an initial interpreter or parts of such a program.

Upon Berkeley's initiative, Lawrow was then invited to the USA, officially as a guest of Berkeley's company. Here, Lawrow could visit MIT, learn about LISP, and receive information about its implementation.

In 1968, after Lawrow's return to Moscow, the actual programming work began at the Academy's Computing Center in Moscow. The system, in its initial version, was running by 1968, and it was presumably stable since early 1969 [LAW69]. G.S. Silagadse, a Georgian, joined Lawrow during the implementation. Under Lawrow's supervision, he ensured the correctness of the system and began working on its compiler.

Around 1971, Silagadse presented a doctoral thesis on the LISP compiler [SIL71a]. He took over the maintenance of the BESM-6 LISP independently after Lawrow moved to the University in Leningrad.

When Silagadse returned to Tbilisi around 1973, W.M. Jufa became the custodian of the LISP system. Under his direction, several new features were introduced in 1973, and the system was integrated into the Time-Sharing System PULT, developed by the Academy Computing Center [BRI72]. However, the limitation of the user space to 32K BESM-6 words (48-bit) was always a serious constraint on the complexity of potential applications. It's no wonder that LISP was not seen as a future basis for research for a long time. The advent of ESER computers with their relatively large main memories may have changed the situation.

Lawrow continued his work in Leningrad. The basic system for this was a LISP 1.5 system implemented on an Odra 1204 in 1973.

Since around 1974-75, possibly significantly stimulated by the 4th International Congress on Artificial Intelligence in Tbilisi [AD75], much livelier activity in the field of Artificial Intelligence has begun in the USSR. This was accompanied by the development of several LISP systems.

BESM-6 LISP, which was the only widely used system until 1973, has since acquired several competitors: ?. Pantelejew developed a system in 1975 on the ICL-4 that was compatible in every respect (instruction by instruction) with BESM-6 LISP. ?. Gitman, ?. Feinman, and some colleagues began implementation for ESER computers in Vladivostok in the same year. They continued this work in Leningrad with Lawrow and developed an interesting, fast system that runs on OS/MVT and corresponds to LISP1.6 in many respects.

In Moscow, the LISP/EC system was developed in 1975 at MEI. This system operates in DOS. It corresponds to neither LISP 1.5 nor other models and includes both common functions and singular decisions [BAI77]. Noteworthy is the direct programmability of the coupling of LISP and PL/I programs, which is accomplished through the CALL function. The system offers data structures such as strings and arrays. In 1979, LISP/EC was ported to OS.

In addition to their own developments, some foreign LISP systems have been available in the USSR for quite some time. This is true for STANFORD LISP/360, which found its way through Dubna, for UT-LISP, which runs at least on the CDC 6600 in Dubna, and for DOS/ES LISP1.6, which was imported

from the GDR to Odessa at the end of 1976 [GER77]. When McCarthy visited Novosibirsk in 1968, he started implementing a LISP system for the BESM 6 using the ALPHA systems programming language. This system was later continued with Russian function names and was completed only in 1975-76 by L. Gorodnjaja.

5.10 Hungary

In Hungary, a group of researchers at the SZTTAKI Institute, led by I. Bach, began working on LISP implementation in 1972. The group, including E. Farkas, M. Naszodi, and J. Fabou, was particularly fascinated by the external appearance of the language: the complete distinctiveness from other programming languages and its beautiful form captivated these scientists who had been exploring Artificial Intelligence between 1970 and 1971.

They were aware of Deutsch's pdp-1-LISP, which served as a model, and LISP F1 from Uppsala.

By 1973, an interpreter was implemented, encompassing about 40 functions, and it ran on a simple time-sharing system on the R10, a Hungarian development. The list memory capacity was around 4K.

In 1974, F. Potari at TKI, also in Budapest, began implementing a larger system for the same computer. The system was intended for circuit design. Potari programmed over 180 functions, developed a file processing system, and attempted to integrate a rule-based subsystem into the LISP system.

The system only allowed whole numbers. For the notation of atomic symbols, 72-character long names were allowed, and beyond that, each character could appear as a character atom if it was enclosed in ". One-dimensional arrays, called vectors, and associated basic operations were also present.

For rule-based programming, Potari introduced the LAMMAT expression: It took the form (LAMMAT modvarlist (form rulelist)). Instead of form, in certain cases, only a function name could stand. The rules took the form: (model predicate value). The pattern functions were passed through EVAL as EXPR functions with four arguments: the Modvar list (pattern variables from the rule list), the rule list, the arguments (as a list), and the current A-list.

The implemented function EVMATCH processed rules from the rule list using the following approach: First, it tried whether the argument list matched the pattern (model). If successful, the pattern variables were bound, an A-list extended by these bindings was created, and predicate was evaluated. If this evaluation returned T, then value was evaluated in the same binding environment. This determined the overall value of the LAMMAT expression. If either the pattern match or the predicate evaluation failed, it moved on to the next rule.

The entire system was connected to an external library storage that could be used automatically or selectively. When all standard functions were loaded, only space for 2000 list cells remained. With the help of a swapper to the hard head disk, large programs could also be interpreted in this environment. However,

Potari noted in his description [POT75] that 64K of free memory would be required for reasonable applications.

After developing a compiler in 1976, Potari had to abandon his system. His users had concluded that lists were a poor representation, and the system was too inefficient overall.

In 1977, at the SZAMKI Institute, P. Szöves began a LISP implementation for a Honeywell H80. He aimed to create a system compatible with InterLISP.

5.11 Italy

In Italy, LISP has been in use since at least 1966. For instance, the Laboratori Nazionali di Frascati in Frascati near Rome is mentioned ([CAM66], p.63), where physicists apparently utilized LISP for symbolic manipulation purposes. Shortly afterward, A. Miola, along with friends (C. Böhm, G. Ausiello, H. Moncayo, L. P. Deutsch), implemented LISP on an Olivetti prototype CINAC, which his institute (LAC-CNR in Rome) had received as a gift. The work was highly complicated because practically no program ran on the machine, and they had to start from machine code.

Since around 1972, research on Artificial Intelligence problems has been conducted both at the University of Pisa and at the Institute for Information Processing. While Italy had previously managed with imported systems, there was a belief that they needed to develop their processing system for LISP.

The first LISP system was built in 1972 for the IBM System/360 [ASI72]. It was considered more as a base system for PLANNER or CONNIVER-like extensions. In this direction, some attempts were made in 1973 [MON73a, MON73b]. Parts of these systems were likely implemented for experimentation. In Pisa, higher-level programming languages like PL/I and FORTRAN were regularly used for this purpose.

All these efforts culminated in 1975 with the development and implementation of MAGMA-LISP [MON75]. With this system, they aimed to provide a "machine language for Artificial Intelligence" that is usable for implementation purposes and offers all basic operations. The fundamental concept in MAGMA-LISP is the Context, an information structure containing both the data or facts of a specific situation in the computation process (i.e., the state of the data basis) and the information crucial for dynamic execution, i.e., relationships to comprehensive processes and return points.

The Contexts are interconnected in a tree-like structure. The user can manipulate this tree structure or explore it at any point during processing. Using the APPLY function, one can initiate processes within certain Contexts. In this approach, portions of the information present in the original Context are simultaneously transferred to the new Context. The exchange of Context components can also be directly triggered through the GET and PUT functions.

By the end of 1975, MAGMA-LISP was operational [ASI75]. It had been implemented in FORTRAN and ran on the IBM System/370-168 under the VM/370-CMS operating system. It was used for education and research in Pisa for several years.

5.12 Denmark

In Denmark, it seems that LISP remained unknown for a long time. Perhaps various facilities used foreign systems. An indication of non-use is that there was no LISP system available on the University of Aarhus's CDC 6400 as late as 1975.

In the spring of 1975, as part of a lecture on "higher-level programming languages," a LISP implementation was initiated. Twenty-five students, under the guidance of N. Derrett and M. Solomon, programmed eleven small modules in BCPL, creating an interpreter. The decision was made not to include PROG and to use simpler access scope rules than in LISP. The dialect was named "Lisbet." The same action was repeated in the spring of 1976. Finally, in 1976, one of the two leaders modified the program to make it significantly machine-independent. It can now run on any computer with a word length of 16 bits or more and a BCPL compiler [DERR77].

5.13 Czechoslovakia (CSSR)

LISP activities in Czechoslovakia began after a visit by Konicek from the Prague Technical University to Stanford around 1970. K. Müller and J. Kolar started developing a LISP system for the Tesla 200 at the end of 1971. It closely adhered to the description of LISP 1.5 [MCC62c]. Except for the compiler and the arrays, all elements were faithfully implemented. This system was operational around 1972 [KOL74] and has been used for teaching since then.

Another, smaller system was implemented as part of a thesis at the VUMS Institute.

In the meantime, STANFORD LISP/360, along with the symbolic manipulation system REDUCE, is available in Czechoslovakia.

5.14 Belgium

In Belgium, as in other Western European countries, LISP has been known since the early 1960s. Systems on the IBM 7090 were certainly used. A local implementation was created by P. L. Wodon in 1967 at the MBLE Institute in Brussels [WOD66]. The publication of the first LISP textbook in French by D. Ribbens [RIB69], currently at the University of Liege, also suggests LISP activities: referring to his system programmed in 1967 at the Faculté Polytechnique in Mons for the Bull Gamma M40 [RIB67].

5.15 Switzerland

In Switzerland, energetic LISP activity began with the establishment of the Institute for the Study of Semantics and Thinking (ISSCO) in Castagnola near Lugano. A powerful system was needed for natural language analysis. As they had access to a CDC 6600, they adopted the current version of UT-LISP (circa 1974). However, researchers, including N. Goldman and P. Hayes, found the

version insufficient and devised numerous improvements [GOLM75]. Unfortunately, the new LISP system could not inspire new research results as the institute had to close at the end of 1975 due to financial constraints.

Since around 1976, InterLISP/360-370 has been available at ETH Zurich and the University of Zurich [EP77].

6 LISP Overseas

6.1 Mexico

H. McIntosh established another LISP center in Mexico City. In the context of American LISP history, it was mentioned that McIntosh moved to Mexico City in 1963. Shortly after him, L. Hawkinson also arrived there.

The work in Mexico was conducted either at the Centro Nacional de Cálculo or at the Nacional Polytechnical Institucio. An IBM 709 was available there, and around 1964, Hawkinson implemented a LISP 1.5 system on it.

In 1965, A. Guzman started his work for CONVERT. According to his statements, he could use the Q-32-LISP, reachable via remote processing, in Santa Monica, and two Mexican LISP systems. Hawkinson and Yates worked on these local systems [GUZ67].

The faster of these two systems was named MB-LISP. It also ran on the IBM 709. It was described as "a LISP dialect that differs from LISP1.5 in a whole series of technical details..." It offered "excellent array and floating-point capabilities" and was overall "carefully organized"[?].

Around the same time, the physicist T.A. Brody, working in Mexico, convinced himself of the utility of LISP for symbolic calculations in physics. He developed his own version for an IBM 1620, the LISPITO [BROD65].

Between 1967 and 1971, little is known about LISP activity in Mexico. However, the implementation of LISP 1.6 for the B6700 computer [MAG74] demonstrates that there was ongoing work.

This system was developed in 1971 by M. Magidin and R. Segovia at the Centro de Investigación en Matemáticas Aplicadas y en Sistemas (or at the University of Mexico City). B6700-LISP is programmed in Burrough EXTENDED ALGOL [BURR64]. It could operate either in batch mode or interactively.

The user encountered EVALQUOTE, so pairs of S-expressions were processed. The list storage was a field of 64K, in which spaces (i.e., pages) of 256 words were type-separated into the following data types: lists, atoms, large integers (small ones represented as pointers), and arrays. The LISP cells were packed into 48-bit words, so both Car and Cdr pointers were 22 bits long. Atoms comprised two words for value, P-list, and processing information, as well as additional words and the densely packed area of element pointers. Large arrays were broken down.

The B6700 LISP had about 140 basic functions in 1974, many of which were more reminiscent of InterLISP than LISP 1.6. An editor was available for interactive work.

6.2 Japan

In Japan, extremely lively LISP activities have been observed since 1974.

Until around 1967, no works are known. It may be that LISP 1.5 was available on imported systems before that. Around 1967, apparently the first Japanese LISP system HELP was implemented on a HITAC-5020 in a collaborative effort between the University of Tokyo and the computer company Hitachi. Details are not available.

In 1968, M. Nakanishi from Keio University (Keio Institute of Information Science) followed with his system KLISP [NAK68]. This system, intended for educational purposes, simplified LISP in various ways. It had only one interpreter and about 120 basic functions. The base computer was a TOSBAC-3400/30.

Afterward, Nakanishi became acquainted with the pdp-8 implementation by van der Mey and van der Poel. After consolidating the experiences with KLISP and incorporating ideas from Delft, he, with the help of colleagues, programmed the systems MINILISP and KLISP-11 for the small computers HITAC-10 and pdp-11, respectively [NAK69].

By 1970, there must have been significant results and experiences, as evidenced by a compilation edited by H. Ammiya [AMA70].

Before 1974, the systems DOS/EDOS-MSO LISP 1.5 (Watanabe and Futamura, Hitachi Ltd.) on a HITAC-8350 and TLISP (Miyazaki et al.) on TOSBAC-5600, which was inspired by KLISP, emerged.

In 1972, Toshiba Corporation's Electrical Engineering Laboratory started implementing an internationally comparable system. The decision was made to adhere to LISP 1.6. This system, EPICS-LISP 1.6, primarily constructed by T. Kurokawa, ran on the Tosbac-56000 and replaced TLISP. By early 1973, a manual was drafted. Kurokawa continued working, and a year later, he believed that the name LISP 1.6 no longer represented the system adequately. Thus, the manual [KUR74] was overwritten with "LISP 1.8."

Using ideas from InterLISP, Kurokawa subsequently modified the memory. In particular, the representation of data structures was changed to maximize processing speed [KUR76c]. His ideas on a more efficient garbage collection, which in 1975 still revolved around stack memory utilization within the simple McCarthy algorithm [KUR75], matured into an improvement of the basic algorithm and addressing stack overflow issues [KUR76b]. This careful analysis of the critical parts of the interpreter proved successful: once the weaknesses were addressed, LISP 1.9, as it was now called, achieved significant runtime improvements [KUR76c].

By 1979, Kurokawa's LISP 1.9 was likely one of the most efficient systems in the world. It was certainly the fastest with conventional memory organization [IPSJ].

In 1973, Sato et al. proposed a new hash scheme for LISP, inspired by earlier ideas from McCarthy that had influenced the incomplete M44-LISP by Wooldridge. Goto embraced this idea, suggesting that only specific list structures should be subject to the hash scheme. His system HLISP [GOT74a],

based on this idea, consequently introduced two CONS functions: CONS and HCONS. Later, he further separated LISP structures from those subjected to hash management, calling them "tuples" and "sets" [GOT76].

The first HLISP system ran in early 1974. In addition to memory management, this system was equipped with a result management, another idea by E. Goto: he noticed that in many recursive calculations, previously evaluated expressions were repeatedly processed. This is especially true for many base cases of recursions. However, it often happens that more complex expressions are also calculated multiple times. Thus, a significant portion of computing time is wasted determining previously known results.

Goto equipped his HLISP with the Assocomp feature, where values of previously specified functions are stored and automatically retrieved with associative mechanisms whenever possible. Thus, the ordinary evaluation mechanism, supplemented by table lookup, became significantly more efficient.

As the base system of the REDUCE formula manipulation system, HLISP helped achieve substantial reductions in computation time [GOT76].

In 1974, the Japan Society for Information Processing organized a Symposium on Symbol Manipulation. On this occasion, a comparison of operational LISP systems was arranged, the 1st LISP Contest. Ten different systems running on ten different machines participated. In addition to the aforementioned systems HLISP (then on HITAC-5020), EDOS/EDOS-MSO LISP 1.5, KLISP, and EPICS-LISP 1.6, the test programs also ran on the systems OLISP, LISP 43, and T-40 LISP. Unfortunately, the two remaining systems are not documented.

OLISP was implemented by H. Yauis and colleagues at Osaka University. It was a relatively small but quite efficient system. The base computer was called NEAC-2200. After 1974, the system, now on a NEAC 800-2, was equipped with a compiler. It largely adhered to LISP 1.5 but, with its limited user base, retained local significance.

LISP 43 was created by Y. Yoshida from Nagoya University for an OKITAC-4300. It was considered LISP for mini-computers. The same applies to T-40 LISP at Kyoto University, programmed by K. Nakamura for a TOSBAC-40C.

A larger system was LISP/M implemented by S. Ono and colleagues at the Mitsubishi Electric Laboratory on a MELCOM-1000, which also had a compiler.

Since 1974, interest in LISP in Japan has become extremely lively. Goto's introduction to LISP [GOT74b] may have contributed to this. While systems like HLISP, LISP 1.6, and OLISP continued to evolve, new implementations were undertaken: in 1974, LISP 1.5 by H. Kimura at the Okayama University of Science on the MELCOM COSMO-700, closely related to the original LISP 1.5; in 1975, the systems NIT-LISP-U (T. Niwa et al., Nagoya Institute of Technology) and VS-DLISP 1.5 (H. Imura, Doshisha University); and LIPQ [TA75].

In 1976, six new implementations emerged (ALPS/I, HLISP-Compiler, LISP 38, PETL, TLICS, and LISP/P). In 1977, FLISP was added, and in 1978, HITAC-10 LISP and LISP 1.7. Most of these systems are also used at the original university, often only by the implementors. Only HLISP on the HITAC

8800 or 8700 and TOSBAC LISP 1.9 have been able to attract larger user communities.

In 1977, the Interest Group for Symbol Manipulation and Natural Language Processing was founded. By the end of 1977, this group prepared a new comparison, the LISP Contest 1978 [IPSL]. This time, in addition to 17 Japanese LISP implementations, three foreign ones participated: UT-LISP on CDC6600, STANFORD LISP on B1726, and DOS/ES LISP1.6. It turned out that many Japanese systems had efficiently realized the list processing of LISP1.5. However, it was not clear to what extent they adhered to the support standards of American systems or had appropriate function spectra.

In Japan, LISP is apparently widely used for educational purposes. In addition, it is applied in the usual areas: formula manipulation, language processing, program synthesis and analysis, robotics, and other fields of artificial intelligence. Recently, there has been a lot of activity in Japan in developing LISP machines.

6.3 Canada and Australia

With regard to Canada, the development of LISP/360 at the University of Waterloo by Kent has already been mentioned, which was completed by J.F. Bolce and R.H. Cooper [BOL68]. No further independent implementations seem to have been carried out, as new systems were constantly available from the neighboring United States.

J.A. Campbell brought a very early version of UT-LISP (see p. 232) and Kent's 3600-LISP (see p. 243) to Australia in 1967. The latter ran on a CDC 3600 in Canberra. Campbell brought the CDC 6600 up to the state that he had previously achieved with his LISP 1.6 at the Imperial College (see p. 236). It was used in Adelaide. Apart from this, only activities with other list processing languages, for example by J. Hext using WISP (see p. 236), are known.

6.4 India

In India, LISP has been used since the introduction of IBM 7090-type machines or similar. One domestic implementation is known: that of N. Soundarajan for the TDC-316, an Indian computer.

7 Metasystems

7.1 Classification

A brief overview of LISP history cannot be complete without considering the metasystems built on LISP implementations.[20] These systems, or the languages realized through them, have often served as the means through which users interacted with the LISP system.

[20] Actually, a chronological presentation of applications would also belong here.

These metasystems are closed language processing systems implemented based on LISP (usually with LISP itself as the system programming language) and often differ significantly in syntax from LISP. Considerable influences from ALGOL have been common, but programming languages for linguistic problems (such as COMIT) have also served as models. Improvements over LISP are typically characterized by language elements intended to make programs more understandable,[21] and those that enable new processing methods. The latter are often reactions to newly developed methods in artificial intelligence research.

Early on, it was recognized that while LISP is unparalleled in the simplicity (sometimes called elegance) of its syntactic structures, it can be challenging to read and understand for the development of complex programs by humans, especially for newcomers to programming. These programs lack connections to the notation methods familiar to the normally educated programmer.

The flow of programs is often hard to discern; within PROG blocks, undisciplined programming styles, as in the most basic FORTRAN, are possible, and free work with global variables and functional arguments creates many comprehension difficulties. On the other hand, there is no doubt that such language elements contribute significantly to the expressive power of LISP.

These disadvantages of LISP became evident when compared with ALGOL. Already in 1961, thoughts turned to a subsequent language version. The ALGOL model hinted at the utility of clearly defined, block-structured variable scopes for cycle organization, language elements suitable for cycles, and improved readability of expressions with fewer parentheses (by introducing different parentheses for mathematical expressions and program structures, as well as precedence rules).

7.2 Before LISP 2: Pattern Matching and System-Independent Input

Due to the specific application areas of LISP, attention was early on directed to languages designed to address problems of string processing. One of these, COMIT, was designed around the same time as LISP at MIT [YN61,YN62,YN63a,YN63b]. What was highly interesting about this language was that the author Yngve had translated a grammar description based on Chomskyan principles into a programming language. To describe language structures, sequences of terminals and non-terminals were noted—structured by separators (+). The non-terminals acted like variables that could be used for entire classes of character strings. Newell and Simon had also worked with patterns of formulas when proving logical statements to determine a possible conversion by substitution. The diverse use made pattern matching of character strings or formula structures an important method of processing non-numerical data.

Bobrow, who was involved in natural language processing, wanted to have

[21] What is understandable cannot be easily decided. In the case of LISP, these were often forms from other languages.

these capabilities in LISP and built his system METEOR on LISP 1.5 [BO63a]. METEOR is, in a sense, the first metasystem over LISP. The language elements are rules written as LISP lists. The language itself is activated by calling the interpreter (METEOR) and passing rule sets and data as arguments. For the rules, some special characters are used as METEOR syntax elements: $, and / play special roles. The rules themselves consist of seven parts: rule name, flow information, pattern (left side), transformation directive (right side), section for managing temporary storage, name of the next rule, comment. All these elements are either real atoms or lists.

Rule execution usually follows this order: an attempt is made to match the pattern to the processed character string.[22] If successful, the reordering and storage directives are executed, and the next rule is jumped to as indicated. If the comparison fails, the next rule is processed.

Patterns are described by literal atoms, numbers, and dollar elements. The linearity of data is expressed in the fact that, after pattern matching, individual parts must be addressed by numbers. If a number appears in the pattern, it means the repetition of the corresponding element. With $ expressions of the form ($.N), consecutive elements of the working memory can be referenced. By $0, an initial or final position can be specified, and ($.ATOM) and similar forms can determine data types.

Overall, METEOR is certainly not a fully developed metasystem, as it clearly adheres to LISP syntax. It does not use its own reader routine, especially. With the inclusion of a processing style foreign to LISP and the introduction of pattern matching, METEOR triggered subsequent effects.

METEOR Example Program:

```
(METEOR
(
 (CHANGE  ($ ROSE) (FLOWER)        (/ (*Q SHELF1  1 PRETTY))        CHANGE)
 (*       ($)      ((*A SHELF1) 1) (/ (*D PNTRET RULE3))           *)
 (PRNTWS *   ((*P THE WORKSPACE IS))                               PNTRET)}
 (RULE2   ($)                                                      END)
 (RULE3   (($.1) ($.1)) 0 (/ (*S ODD 1) (*Q EVEN 2) (*D PNTRET RULE3))}
                      PRNTWS  (THIS IS A CONTINUATION OF THE PREVIOUS CARD))
 (*       ($)      ((*A ODD) (*N EVEN)) (/ (*Q ODD (*N EVEN) ONLY)
                                          (*P ODD EVEN) (*D PNTRET RULE2)) PRNTWS))
((A ROSE IS A ROSE IS A ROSE))})
```

Fig. 10. An example METEOR program [BO63a][Fig.4]

In the course of 1964, a shift was made to processing syntax foreign to LISP. J.Hext is likely to have been the first to use the LISP 1.5 basic functions for character-by-character input. However, Hext did not construct a metasystem but developed only an input part for converting arithmetic expressions into LISP

[22] Pattern Matching includes both comparison and identification of certain parts and their storage.

S-expressions. MATHREAD [HEX64a], as the main function was called, could, however, serve as a subroutine for a metasystem. As such, this program was an essential part of the first REDUCE system by Hearn, as he had requested the program from Hext.

Based on the experiences with MATHREAD, Hext formulated general methods for translating any programming language into the LISP format [HEX64b]. However, the syntax description and the translation procedure do not seem to have been incorporated into a LISP program for a compiler.

7.3 LISP 2

While programs for formula manipulation purposes were slowly developed during this time, the first major project of a metasystem entered the phase of realization: LISP 2. Although conceived as a successor and thus standing at the same level as LISP 1.5, LISP 2 primarily emerged as a metasystem and, as such, faded away.

As the indefinite discussions about "the new language" became concrete, other programming languages had emerged by then and seemed worth considering. Among these, SLIP [WEIZ63] should be mentioned, like FLPL, a list processing language built on FORTRAN, consisting of a set of subfunctions (mostly written in FORTRAN) appended to FORTRAN. SNOBOL [GRIE68], a language still used for string processing [FAR64], included many changes and improvements regarding COMIT. The enhanced means for describing patterns and influencing the comparison process seemed highly significant. Also, the language TREET [HAI65] opened new perspectives. It was based on LISP[23] and used ALGOL- or FORTRAN-like notation for arithmetic and program control. LISP was extended with a set of functions advantageous for working with trees.

The development of LISP 2 may have become problematic towards the end because it fell into an extremely fertile phase of computer science, with so many new ideas expressed that they could hardly be fully thought out.

Work on LISP 2 began in earnest in 1964 when the LISP 1.5 implementation was running on the Q-32 at SDC (see Chapter 4). Employees of Information International Inc. and Systems Development Corporation were responsible for defining the language scope and implementation, with the Q-32 serving as the base computer. The team received advice from scientists from Stanford and MIT.

From MIT, suggestions for string processing seem to have come mainly. The 1964-1965 annual report presented the MIT perspective as follows: "LISP 2 will have dynamic memory management, recursive functions, and dynamic arrays. LISP 2 will include list processing like the current LISP 1.5, but the control language will be more ALGOL than LISP. The used list structures will be compacted to make room for the provision of arrays and new list elements.

LISP 2 will include format-directed string processing, similar to the COMIT programming language. Within this processing, the user can specify a pattern

[23] see [SAM66; O68]

with which a string will be compared. If the comparison is successful, the string will be broken down into parts corresponding to the various subpatterns...

The compiler for LISP 2 will be created, starting from a bootstrap compiler written in LISP 1.5. The actual compiler will be almost entirely programmed in LISP 2, making LISP 2 quite independent of machine type. We expect it to be built on the following machine versions: Q-32 at SDC, IBM/360, DEC pdp-6, GE645... A first version will soon be operational on the Q-32 at SDC." [PROM2].

Notable contributions from MIT include Bobrow's memos on string processing [BO64b] and pattern description [BO65], as well as Levin's proposal for syntax [LEV63]. After Bobrow moved to BBN (early 1965) and delivered the pdp-6 along with the implementation of the new LISP system for this machine, little concrete partial achievements from MIT are known. They had enough to do with themselves, and the implementation of LISP 2 seemed to be on the right track.

Stanford also made many initial contributions: McCarthy thought about basic data structures and basic operations on them [MCC63f]. In August 1964, a proposal for a language definition was even made [MIT64]. The content of this position paper is practically an extension of ALGOL 60. The required new elements were seen not only in the field of list processing (data structures, operators, and standard functions, as well as adapted cycle constructions) but also in the area of language extensibility. New data types, operators, and syntactic constructions should be definable.

Finally, a change to ALGOL 60 semantics was proposed, eliminating call-by-name parameter passing.

Although Stanford is very close to Santa Monica, direct contributions seem to have been lacking in this case since early 1965. It is not clear whether this indicates better proposals from the implementation team or disagreements.

Considering all the proposals mentioned from MIT and Stanford, one can already appreciate the overwhelming task that Abraham's team had to accomplish in realizing unreal visions.

By 1966, the implementation had progressed to the point that a significant report was made in the fall to the FJCC [AB66b]. According to this report, LISP 2 was to be characterized by two language levels: the source language, the actual LISP 2, was ALGOL-like, and the intermediate language corresponded to LISP 1.5. By introducing new data types (arrays, strings, packed data tables) and new operations (subword extraction and insertion and pattern-controlled data processing), a truly viable system was to be created. In 1966, not everything was fully implemented yet, but the essential parts were already well recognizable.

LISP 2 allowed the following data types: Boolean values, integers, octal numbers, real numbers, functional elements, symbols, and arrays. The functional elements corresponded to LISP functions, which could appear as both arguments and values of a function. The "Symbol" data type can be associated with the S-expressions of LISP 1.5. The elements of the arrays could be of any other type, except arrays (arrays).

Expressions (programs) were defined similarly to LISP: they consisted of

constants and variables. Quoting was done with a preceding apostrophe (possibly an invention of LISP 2). Three types of variables were distinguished: Fluid, Own, and Lexical. Fluid variables corresponded in a certain sense to the dynamic variables of the SPECIAL type[24], Own variables served for cooperation with LISP 2-foreign programs, and lexical variables were assigned to ordinary local variables.

To express function applications on arguments, operational forms were used. The function name stood before the argument list, as in ALGOL. There was the option to work with MACRO. Infix and prefix operators for all possible types of operations were allowed. So, for CONS, Mitchell's proposal to use the dot as an operator was adopted, but no new names were chosen for the CAR and CDR operations.

Block structures were adopted from ALGOL. Declarations had to be made (including optional initial values) at the beginning of the block. A range of statement types was provided to structure the program text: CASE, FOR, GO, RETURN, TRY, conditional, compound, and block statements, as well as conditional and block expressions.

A new element here is the TRY statement, which serves to set up processing levels (similar to ERRORSET in LISP 1.5): If an EXIT condition occurs in an expression or statement inside the TRY statement, control is returned to the TRY statement.

Efforts were made to create favorable file processing capabilities for input and output. Similar to LISP 1.5, LISP 2 was also envisioned as interactive, so that only one input file and one output file could be active at a time. By simply switching the input and output assignments, any physical device could be operated. With this flexibility, LISP 2 was a model for the later systems LISP1.6 and BBN-LISP.

The LISP 2 programming system was characterized by the coexistence of the LISP 2 syntax translator, which formed the internal (LISP 1.5) S-expressions from the LISP 2 source language, and the LISP 2 processing system. The LISP 2-LISP 1.5 converter was generated with the help of a compiler-compiler. It was operated by a scanner that worked as a finite automaton.

The peculiarity of the function handling in the LISP 2 system consisted of the immediate compilation of the read functions and followed the Q-32-LISP 1.5 in this regard [SAU64b]. The compiler, as usual in LISP, was divided into the actual compiler and the assembler program and included a series of optimization stages. A mixture of cellar arrangement of numbers and variables, as well as dynamic list storage with compacting garbage collection, was used in memory management.

The LISP 2 team consisted of proven specialists in LISP implementation. In addition to Abrahams, Levin, Saunders, S.Kameny, C.Weissman (author of the LISP 1.5 Primer [WEIS67]), as well as J.A.Barnett, E.Book, D.Firth, and Hawkinson should be mentioned.

By mid-1966, substantial parts of the documentation were already available,

[24] that is, bound by a function and free in subfunctions

including a primer written by Levin [LEV66] and the reference manual authored by Abrahams [AB66a].

The initial successes seemed very promising, but then an unforeseen complication arose: Although the implementation was foreseen on other computers, the language was to be popularized with the system built on the Q-32 first. Due to the remote access to the Q-32, this was not an obstacle. However, it was now replaced by an IBM/360.50.

So, without serious users in sight, work on the second implementation had already begun. By early 1967, all important decisions had been made, a series of descriptions were completed [BAR67a, b, c, d, FIR67, WIL67], and the implementation was in full swing.

In addition to the LISP 2 project, SDC was also working on the ADEPT Time-Sharing System [LI69] for the IBM/360.50. Perhaps it was believed at the time that they could outperform IBM with an operating system and a general-purpose programming language. However, apparently, there was not enough money to sustain LISP 2, which had meanwhile fallen prey to the PL/I disease of "feature cancer" ("More, more, more...!").

Despite the many premature accolades and high expectations, LISP 2 did not turn out to be the great success that the employees and probably the participating companies had hoped for. Since 1968, there is practically no mention of this system, which must have suffered a sudden decline. Only J.Sammet reports in her book on the history of programming languages, which was only published in 1969, still with more than praising words about the LISP 2 project. Sparse voices from the field of Artificial Intelligence later commented: "The aborted LISP 2 system..." [FEN71] or "Unfortunately, they developed it to death..." [EAR73].

Important insights into the causes of the muted reception were provided by J. Weizenbaum in his review of the LISP 2 publication in 1967 [WEIZ67]: In his opinion, the authors of LISP 2 had abandoned essential features of LISP 1.5 without being able to demonstrate correspondingly important advantages. Since most of the possibilities provided by LISP 2 could be built in many other programming languages, such as PL/1 (including pattern-driven data manipulation), there was no longer much reason for LISP 2. Weizenbaum named the "simplicity with which programs can be treated as data (an actively running program can be processed, and a resulting structure can be treated as a program again)" as important and unique features of LISP 1.5 [WEIZ67]. Additionally, Weizenbaum criticized the loss of interpretation as a deterioration of the atmosphere in program development and compacting garbage collection. He predicted a duration of 200 seconds for the latter on computers the size of 200,000 words, which he found unbearable.

Apparently, none of the scientists involved in the discussion about the goals of the LISP 2 project had thought of these features of LISP 1.5. In this respect, the LISP 2 project, through its sudden end, provided important insights into the more accidentally created LISP 1.5 language.

Abrahams mentioned the pursuit of the project deviating from substantive questions and the faulty focus on efficiency: "As the project developed, it be-

came more ambitious, and a large part of the effort was spent on optimization. The biggest mistake of the project was trying to make the first implementation of LISP 2 the best one, instead of just delivering an interpreter to secure the project. There were already enough difficulties implementing the system on the Q-32, and the project was further confused by an attempt to switch to the IBM/360. The project finally collapsed because it consumed resources that were out of proportion to the slow progress. We tried to achieve too much and did it in the wrong order..." [AB76].

Hawkinson sees additional reasons for the failure, on the one hand, in the fact that LISP 2 did not sufficiently perceive the tendency toward interactive use of LISP. This simply had not been a priority in the design, and that was a serious mistake. On the other hand, LISP had increasingly become an implementation language for other systems and was less used to write application programs. "Beautiful syntax and diverse data types are simply not important for implementing a higher-level programming language." The singular machine and the fact that "no 'important' LISP expert" was involved anymore also played a role...

It seems that the negative outcome of the LISP 2 project was one of the most significant events in the field of Artificial Intelligence and perhaps in the programming language sector (comparable to the end of ALGOL 68) in the 1960s. The failure to analyze this event probably contributed to further comparable failures. Many of the processes observed in recent years around Common LISP are very similar to those of that time. The common cause is the lack of understanding of the characteristics that make a good programming language. At that time, it was LISP 1.5 that was not well understood; today, the many mostly arbitrary innovations and extensions of the 1970s remain undigested. So, not much more can be said than: LISP 2 came too early...

```
FUNCTION LCS(L1,L2); SYMBOL L1, L2; BEGIN SYMBOL X,Y, BEST ← NIL;
INTEGER K ← 0; N, LX ← LENGTH(L2); FOR X ON L1 WHILE LX > K DO BEGIN
INTEGER LY ← LENGTH (L2); FOR Y ON L2 WHILE LY > K DO BEGIN ← COMSEGL
(X,Y); IF N <= K THEN GO A; K ← N; BEST ← COMSEG(X,Y); A: LY ← LY-1
END LX ← LX-1 END; RETURN BEST; END;
```

Fig. 11. A LISP 2 example program. [AB66b][S. 670]

7.4 Between LISP 2 and PLANNER: CONVERT, FLIP, MLISP, and Formula Manipulation Systems

In 1965, almost simultaneously, two systems emerged as developments of METEOR: CONVERT [GUZ66] and FLIP [TE65b]. The intention of these systems was to expand the focus on linear lists (representations of character strings) and enable pattern-matching processes for arbitrary S-expressions. Both systems were embedded in LISP, meaning they were programmed as systems of LISP functions, similar to METEOR. The elements of the pattern language appeared as ordinary arguments to the main function, and special characters were used in their formulation, which played no role in LISP.

CONVERT, developed by Guzman and McIntosh in Mexico City, seems to have been the somewhat bulkier, slower, and more resistant-to-modification sister. However, in terms of formulation means, CONVERT practically achieved the quality of Teitelman's FLIP.

In FLIP, each program step, as a call to the main function FLIP, consists of a pattern match followed by a construction controlled by a format. Patterns and formats have almost the same syntax, with the latter being simpler, as the actual localization process for parts in pattern matching is completed.

The patterns are described with almost the same characters as used in METEOR: in addition to }, $, and /, only = is added. Patterns can be described by $ elements for segments, by literals in the form of arbitrary S-expressions, and by expressions to be evaluated. Sublists serve as patterns pointing to deeper list levels. At any point in the pattern, earlier parts of it can be referenced. Pattern elements are intended for references, allowing referencing the main level of the pattern, the current level, and any intermediate levels. This is what makes the use of structured patterns truly powerful.

With special atoms such as $*, /, etc., quite specific actions could be achieved, such as checking properties of pattern parts that are difficult to express with pattern description means, or the immediate termination of the pattern matching process.

FLIP translated rules consisting of constant elements into inefficient LISP programs. This represented a step beyond mere embedding.

Teitelman himself incorporated FLIP into his comprehensive system PILOT [TE66]. There, it could be used by LISP programs but also served to create a convenient interface for the user. FLIP was also used in the part of PILOT intended for program modification, the editor. PILOT was designed as a testing and modification aid for LISP programs and included, for these purposes, alongside the editor, the ADVISE system (for modifying the entry and exit elements of functions, thus influencing the relationships between functions). Interruption functions like BREAK are also integrated into the overall system (see also [TE65a]).

In an advanced version of FLIP [TE67], some of the rough spots were smoothed out. Instead of numbers, special atoms (beginning with # but followed by the old number) are now used for referencing pattern elements. To carefully distinguish literals from calculable pattern elements, the former are quoted (with '). The numbers following $ (to designate segments of a certain length) or # can be replaced by arbitrary expressions with a numerical value. Predicates can now be directly assigned to sub-patterns or pattern elements. The reference to the assigned element is accomplished through the asterisk $*$.

A shorter pattern formulation is achieved with the help of the keywords EITHER (alternatives of patterns) and REPEAT (repetition). Whole calculations during pattern matching can be achieved with the help of the assignment SET.

Around the same time, several systems for formula manipulation in LISP were developed. In contrast to contemporary program systems for Artificial Intelligence, users of a formula manipulation system seemed to require not LISP notation but rather a more common mathematical notation.

```
(DEFINE ((FORMTRAN(LAMBDA(U)(CONVERT
(LIST)
(QUOTE(X(LL)(RR)))
U
(QUOTE(*(
       ((LLL P RRR)(P(=BEGN=(LLL))(=BEGN=(RRR))) )
       ((LLL M RRR)(M(=BEGN=(LLL))(=BEGN=(RRR))) )
       ((LLL * RRR)(A(=BEGN=(LLL))(=BEGN=(RRR))) )
       ((LLL / RRR)(/(=BEGN=(LLL))(=BEGN=(RRR))) )
       ((LLL ** RRR)(**(=BEGN=(LLL))(=BEGN=(RRR))) )
       ((X)(=BEGN= X))
       (()_0)         )))   )))   ))
```

Fig. 4.12 A CONVERT example program [BO68b][pp. 41-42]

```
(DEFINE((DERIV(LAMBDA(EXPR VARBLE)(TRANSFORM EXPR
      '((==VARBLE 1)
       (\$1[ATOM*] 0)
       ((.. + ..)  (SPLUS DL DR))
       ((.. - ..)  (SMINUS DL DR))
       ((.. * ..)  (SPLUS (STIMES LL DR)(STIMES DL RR)))
       ((.. / ..)  (SDIV(SMINUS(STIMES RR DL)(STIMES LL DR))
                                              (SEXPT RR2)))
       ((.. ^ ..)  (STIME RR (STIMES SEXPT LL (SMINUS RR 1))
                                              DL))))
   NIL
   '(K    PAV  $1[NUMBERP*]
     L    PAV  $1[NUMBERP*]
     LL   SKEL =(UNLIST #1)
     RR   SKEL  =(UNLIST #3)
     DL   SKEL  ( = (UNLIST =/= 1))
        ...
```

Fig. 4.13. Part of a FLIP example program [TE67][p. 84]

While MATHLAB [ENG65] completely ignored the input problem and expected users to translate it into LISP expressions, it did provide readable results [ENG66].

Similar circumstances existed for the "Symbolic Mathematical Laboratory" by Martin [MART67]. Following a suggestion by Minsky [MIN63b], Martin had strived for the most faithful output of mathematical expressions on a screen and ways to edit or modify these expressions with a light pen and keyboard, along with the symbol manipulation system. The original input was minimally treated by Martin.

On the contrary, REDUCE [HEA67], from the beginning (1964) and using Hext's program [HEX64a], solved both the input and output in LISP, providing

a standard mathematical notation. Similarly, SCRATCHPAD [GRI71b] proceeded. REDUCE2 [HEA73], the successor developed between 1968 and 1970, became a true metasystem. The significant innovation was the adoption of an ALGOL-like notation for data and programs, spread over the LISP base. This higher programming language, known as RLISP today, introduced many of the basic operations of LISP as infix operations[25], allowed the notation of parentheses-free expressions following the usual precedence rules, and offered some instructions for structuring programs, among which the FOR statement is notable. MLISP followed suit, being the first true metasystem [EN68].

Even in SHARE-LISP, i.e., in the version of LISP 1.5 for IBM/7090 distributed by Stanford since 1965, there were individual functions for control flow that derived their names from ALGOL keywords ([MCC62c], 1965 edition, p. 98). When the STANFORD LISP/360 became operational at the end of 1967, H.Enea began implementing a preprocessor for the M language. This was intended to revive the old, almost forgotten M expressions and free users from many parentheses.

MLISP introduced infix operators for the essential arithmetic operations. Only for EQUAL and APPEND, as two operations on S-expressions, there were also infix operators (= and @). Prefix operators were +, -, NOT, NULL, ', and ATOM. While any function of two arguments was allowed as an infix operator with its name, this was not the case for prefix operators.

MLISP, like LISP, was an expression language, meaning all language elements were intended with a value. In addition to assignment and conditional expression, some cycle constructions (FOR, UNTIL, WHILE) had been developed, which, in addition to the ordinary ALGOL-like execution, also included a variant (COLLECT) in which the partial results were collected in a result list. A special syntactic construction had been developed for the frequent LISP operation LIST. Finally, function definition was also formulable in MLISP.

By mid-1968, MLISP was operational and was only used as a preprocessor; that is, MLISP programs were translated directly into LISP and could be executed as such.

When the pdp-10 became operational at the end of 1968, a similar system was demanded for LISP 1.6. D.C. Smith implemented MLISP and introduced some extensions: floating-point numbers, character strings, a new form of the FOR expression, and improved error reactions [SM69]. With the growing complexity of language constructions and the now established error level (ERRSET), there was a certain separation of normal LISP processing from the sometimes substantial MLISP interpretation. However, the internal representation of MLISP programs cannot be distinguished from that of the host system LISP.

In the course of 1969, Smith further developed MLISP to the state where it was used for a long time [SM70]. Significant innovations were the vector operations, which, however, especially due to vector assignment, increased the level of interpretation. Vectors could be understood in two ways regarding assignment: one could change an element, as expected; if L is a vector (i.e., a

[25] = for EQUAL, . for CONS

list) of the form (A B (C D) E F), then the assignment

```
L[3,1] ← 1
```

results in the vector L now being A B (1 D) E F.

However, a vector assignment (decomposition assignment) considers the elements of the vector as changeable variables and changes their values depending on the structure:

```
L ← '(1 2 (3 4 5 6 7) (8 9))
```

causes the following changes: A now has the value 1, B has the value 2, C has the value 3, D has the value 4, the piece from 5 to 7 is ignored, E gets the value (8 9), and F, since no element is left, gets the value NIL. It is easy to see that with this assignment, a good part of pattern matching operations can be performed.

The experience with the construction of the syntax analyzer in LISP eventually led Smith and Enea to develop a general schema for translating a language description in BNF into a LISP translator (SDIO), similar to Hext [HEX64b]. The progress of other colleagues, especially at MIT and SRI, eventually led them to MLISP 2.

```
REVERSE2 := #L:
IF NULL THEN NIL ELSE REVERSE2(CDR(L)) @ ;
REVERSE3 := *L:
  BEGIN NEW I, V; SPECIAL I, V;
    FOR I IN L DO V <- I CONS V;
    RETUNR(V);
  END;
REVERSE4 := #L:
  BEGIN NEW V; SPECIAL V;
    WHILE L DO PROG2(V <- CAR(L) CONS V, L <- CDR(L))
    RETURN(V);
    END;
```

Fig. 4.14. An MLISP example program [SM69][p. 29]

7.5 PLANNER: Backtracking, Pattern Matching, and Deductive Mechanisms

The significantly new element that entered the discussion about new means of expression for programming in the field of Artificial Intelligence was the acceptance of backtracking as a useful control principle.

A program uses backtracking when, at certain, not entirely analyzable points, it saves all temporary information and continues working on a trial basis. These points are called decision points. If the program reaches the goal on the chosen path, then the decision was correct, and the saved information is no longer

needed. However, if the goal is not reached, either because a time limit has expired or an error interrupts the processing, then the program restarts at the decision point. Among the saved information, which is used to restore the old state as far as necessary, there are also those related to other possible alternatives.

These alternatives are now tried in a similar way to the first one.

An undesirable state occurs only if all alternatives fail. In this case, one must consider the history: if the decision point was the only one, then the entire program can only be terminated unsuccessfully. However, if there are other decision points before this one, it is treated as if the program, in pursuing the current alternative of the penultimate decision point, has detected an error: the last decision point is lifted, and the penultimate point is retried.

Already in Abraham's proof checker [AB63], this method was applied. For these purposes, the ERRORSET function was invented around 1961, which oversees the restoration of a certain state. The publication by Golomb and Baumert [GOLO65] probably highlighted backtracking as a programming principle for the first time.

With the appearance of this work, the discussion about the methodological use of backtracking had truly gained momentum at MIT, even though it had been used as a programming principle before. Automatic backtracking was then incorporated by Hewitt into the processing model of his language PLANNER.

Initially, PLANNER was considered a theorem prover and then as a tool for controlling robots. Over time, however, it increasingly became a reflection of modernist ideas in the field of Artificial Intelligence and programming methodology.

The first publication on PLANNER appeared in 1967 [HEW67]. After this start, additional ideas were quickly integrated, and by 1969, PLANNER consisted more of a complex of various ideas or subsystems: one part related to pattern matching, another part included the language elements for procedural planning and contained both imperative and declarative elements, another part constituted the pattern-controlled Information Retrieval System, and finally, the execution was controlled by the deductive system and the hierarchical control structure [HEW69].

For pattern matching, a notation called MATCHLESS, summarily referred to as a pattern-based programming language, was used. According to Hewitt's own statements [ibid.], the ideas for this came from the programming language CONVERT [p. 279]; they were further developed through the study of Post's production or general regular expressions.

From CONVERT, Hewitt adopted the use of arbitrary names as pattern variables—unlike the restriction to numbers in COMIT, METEOR, or FLIP. Similar to CONVERT, the type had to be declared: the types Pointer, Atom, Segment, Integer, Floating-point number, S-expression were available.

In contrast to all predecessors but picking up a certain development in FLIP, prefix symbols were used to indicate the different uses of pattern variables. For example, a prefix $ meant that the following variable could be assigned a value during the pattern matching process, $$ meant that the following variable only

fits an object that matches its value, $? meant that the variable may receive a value if it does not have one yet, otherwise, it only matches its value, and so on.

The patterns can be linked by certain functions. These functions do not return a value but only make sense in a comparison process, such as triggered by the function ?assign, where the variables appearing in the pattern receive a value.

The MATCHLESS language also included a formalism to introduce new pattern functions. With all these elements, much more complicated pattern matching processes could be described than was possible until then.

For the debate about whether expressions are declarative or imperative (cf. p. 100ff), Hewitt had a new solution ready in 1969: he considered the duality of imperative and declarative statements as the essential idea of PLANNER. For example, a sentence like `(it follows from A B)` is clearly a declarative sentence, i.e., a statement about a fact. However, this sentence can also be used imperatively in PLANNER, e.g., to introduce a procedure that should check whether A can already be accepted and, if so, initiate the inclusion of B. Moreover, the sentence can lead to the introduction of a procedure that, if the goal is ever to deduce B, suggests deducing A as a subgoal ([HEW69], p. 295).

Thus, the sentence corresponding to the pattern (implies $___A $___B) is used in two imperative forms: as a consequent theorem, which can be used for goal setting (backward chaining: to deduce B, an attempt is made to deduce A), and as an antecedent theorem, which can be used to derive consequences from the fact that a certain sentence has been accepted (forward chaining: B is deduced from A).

When setting goals, the system works with the database. Usually, an attempt is made first to find facts that directly confirm a theorem to be proven (ideally the theorem itself already accepted earlier) or theorems that imply the theorem to be proven.

Since the theorems in PLANNER correspond to procedures, they are activated if they appear suitable for achieving the goal. The programmer creates a system of theorems that, with the PLANNER functions, control the search for usable theorems or facts. Because this search is usually conducted with respect to a goal described by patterns, it is called pattern-directed procedure invocation.

"PLANNER was designed to help in situations where you have a large number of linked procedures (theorems) that could be useful in solving a problem according to a general plan. The language helps to select procedures, improve the plan, and go through the procedures in a flexible way, even if not everything goes exactly according to the original plan..." [HEW69][p. 301].

The functions created for proving theorems and the possibilities to control the direction of proof made PLANNER's logical deduction system more applicable and adaptable than contemporary resolution systems[26].

The hierarchical control mechanism, which is used both by the language itself and is effectively available to the user, is a fundamental basis of the deductive

[26]This does not apply to proof speed.

system. Historically interesting is that Hewitt in [HEW69] hardly emphasizes the backtracking mechanisms but rather emphasizes the strong recursion of the language implementation. It is noteworthy (in this and all later works by Hewitt on PLANNER) that thoughts about possible implementations play a significant role, even though only one actual implementation of a sublanguage was carried out in Hewitt's immediate vicinity.

In 1970, G.J. Sussman and E. Charniak implemented MICROPLANNER for T. Winograd within 14 days, as this part of PLANNER was now called [SUS70]. After initial experiences, this translator was further developed by the three authors with the help of D.V. McDermott, C. Reeve, B. Daniels, G. Benedict, and G. Peskin [SUS72][27]). In [HEW71b], Hewitt speaks of several subsystems that had been implemented at MIT[28]. and that this work was done in MULTICS-LISP and ITS-LISP (i.e., LISP 1.6). However, this is the only reference to other implementations. MICROPLANNER was also adopted into the STANFORD LISP 1.6 and slightly modified [BAUM72].

MICROPLANNER incorporates all essential ideas of PLANNER but is subject to some strong limitations, especially in pattern matching. Analyzing the available descriptions, the following picture emerges: two main objects are used—assertions and theorems. Assertions are ordered tuples of non-numeric objects that can be interpreted as statements about facts. The theorems correspond to procedures in other programming languages; however, they additionally contain an information part that specifies when the theorem should be used and what kind of changes the successfully used theorem induces in the database of assertions.

Both assertions and theorems can be dynamically generated or modified by working programs. As an interpretive language, MICROPLANNER is equipped with several means to structure programs; the interpreter contains parts that handle variable binding, the treatment of state descriptions (control in the normal sense, but also saving and restoring in the case of backtracking). These general capabilities are not directly manipulable by the user. The system not only performs the setup of decision points but also handles the restoration of the environment at these points when the program needs to return there. The user can trigger this process by canceling a program branch.

Compared to the extensive pattern formulation language MATCHLESS in [HEW71c], the description formalism for patterns in MICROPLANNER is strongly restricted. There are only two types of variables (prefix $? and $←)[29], which are only allowed to appear at the top level of the pattern. Patterns are only compared with assertions.

The deductive mechanism uses three types of theorems: consequent, antecedent, and erase theorems. As in PLANNER, the consequent theorems trigger a subgoal setup so that the proof process can be represented as backward chaining. The antecedent theorems aim to derive certain assertions as early as

[27]Also see Hewitt's contribution in [AD73

[28]referring to the language MDL)

[29]$← allows value change, $? one-time value assignment.

possible and include them in the database. For example, if a change in the database by adding new assertions is detected,

```
(THASSERT (HUMAN SOCRATES)) (THASSERT (GREEK SOCRATES)) (THASSERT
(HUMAN NEWTON)) (THCONSE (X) =(FALLIBLE $?X) (THGOAL(HUMAN $?X))) (THPROG
=(X) (THGOAL (FALLIBLE $?X)$T) (THGOAL (GREEK $?X)) (THRETURN $?X))
```

Fig. 15. A MICROPLANNER example program [SUS71][p. 5]

an attempt is made to derive further assertions with the antecedent theorems. In this forward chaining, there is a risk that the database will be inflated with a large number of unnecessary factual statements (assertions).

MICROPLANNER was adopted into many LISP systems. However, there seem to be few actual applications [LAU75]. At about the same time, partly in direct discussion and collaboration, the metasystem QA4 developed.

7.6 QA4

Around 1967, C.C. Green initiated the development of the question-answering system QA1 at Systems Development Corporation. This system was intended to be an advancement of Raphael's SIR [RA63] and was programmed for the Q-32-LISP. Green, along with Yates, continued the work at SRI (where an SDS 940 with BBN-LISP existed). Together, they developed a new version: QA2.

By mid-1969, when Green presented his doctoral thesis at Stanford University, the system had transformed into a resolution prover, incorporating capabilities for working with a database [GRC69a, GRC69b].

This question-answering system then served as a starting point for a new development towards a usable foundation for programming in the field of Artificial Intelligence. The goal was to arrive at a language that should be more natural than first-order predicate calculus and could serve as the background language of a theorem prover, delivering proofs that could be intuitively accepted based on the semantics of the problem. As normal LISP seemed to have too many limitations, a new language was gradually created, named QA4, derived from the original system QA3 [RUL70].

The development of QA4 was initially driven by hand-simulated examples, which came from areas such as automatic program synthesis, robot planning, and theorem proving. Soon, a language was created that differed in some aspects from the syntactic structures of LISP expressions. However, these syntactic differences were later retracted. Alongside decisions regarding the external form, early decisions were also made regarding data types. One consequence of this approach was the determination that all functions in QA4 have only one argument.

The state as of the end of 1970 can be well understood from the authors' contribution to the IFIP Congress [RUL71]. In addition to simple data structures such as numbers, boolean values, and literal atoms, sets, bags, and tuples[30]

[30]Set: each element is represented only once, the order is unimportant. Bag: elements can occur multiple times, the order is unimportant. Tuple: the order is important, i.e., repeated occurrence is significant.

have been introduced. Since a pattern match occurs when functions are called, demanding a bag, for example, allows expressing the commutativity of the function.

Pattern matching during function calls also makes it possible to check the suitability of arguments for specific functions. Through types of pattern variables (e.g., fragment variable, which can be assigned a whole part of an argument) and requirements regarding type or value, as well as by searching for predefined elements in the expression to be checked, argument complexes can be divided into local variables.

Pattern matching is also used for queries to the database, activation of strategies, and inference rules. These strategies are used, for example, to perform a certain parallel execution of programs.

In addition to this parallel execution, QA4 offered the usual sequential execution of instructions, certain iterative language elements for processing data types, and automatic backtracking (triggered, for example, when a function receives inappropriate arguments).

The memory is organized as a discrimination net. Each expression exists only once, as it is reduced to a canonical normal form.

QA4 included a context mechanism before CONNIVER (see p. 298). A context represents a set of variable bindings. Due to the special representation of expressions in QA4, it was possible to assign property lists to them. Thus, the concept of the context includes not only the binding areas of variables but also property areas: an expression can have different properties in different contexts.

Like most other LISP systems and metasystems, QA4 is a collaborative effort of many authors and influenced by discussion partners in various research centers. Besides Deutsch, A. Kay, Sandewall, and a series of users (including R. Fikes, P. Hart, N. Nilsson, I. Greif), Hewitt also contributed many suggestions for further development.

After extensive discussions between the QA4 developers and Hewitt, Sandewall, Kay, and Deutsch, the final version was reached in mid-1972 [RUL72].

The essential features of the system have not changed. However, the system reverted to a source language that is quite similar to normal LISP. Only in the prefix notation for pattern variables are there noticeable syntactic differences from S-expressions.

QA4, like LISP, is an expression language. Expressions are built from simple elements such as names, numbers, truth values, sets, bags, and tuples. Expressions of the form (function Argument) are called applications. Generally, only one argument appears, but for convenience, instead of an argument of type tuple, the sequence of tuple elements can be directly noted.

```
[TURNONLIGHT (LAMBDA (STATUS ←M ON) (PROG (DECLARE N) (EXISTS (TYPE
$M LIGHTSWITCH)) (EXISTS (TYPE ←N BOX)) (GOAL $DO (NEXTTO $N $M))
(GOAL $DO (ON ROBOT BOX1)) ($DELETE (QUOTE (STATUS $M OFF))) (ASSERT
(STATUS $M ON)) ($BUILD ('(:STURNONLIGHT ACTION $M]
```

Fig. 4.16. A QA4 example program. [RUL72][S.295]

The binding mechanism in QA4 is fundamentally controlled by pattern matching. Instead of a variable list, the pattern is specified in which the variables then appear. Corresponding to their function in the pattern matching process, which corresponds to the binding process in LISP, the variables are labeled with prefixes. The prefix $ corresponds to a reference to the value of the variable, the prefix ← to a possible assignment, and the prefix ? represents an intermediate stage: if the variable already has a value, it is taken; if it doesn't, it is assigned. Double prefixes denote fragment variables, which correspond, for example, to a subset instead of a set element or to a tuple piece instead of a tuple element.

A certain set of built-in standard functions and special forms (e.g., PROG) is available. However, the user can define new functions.

The binding operation upon entering a function, which, as mentioned, involves a pattern matching process, can trigger backtracking if the comparison fails. This happens when all binding possibilities are exhausted or if the patterns allow only one possibility. If the conditional operation can be successfully completed or if there are other possibilities, a decision point is set up.

For example, if the following "application" needs to be evaluated:

```
((LAMBDA (SET ←X ←Y ←← Z) (f $x $$z)) (SET 5 6 7))
```

there are nine binding variants, including, among others:

X = 5, Y = 5, Z 0 (SET 6 7) oder
X =5, Y = 6, Z = (SET 7)

Similar to the LAMBDA binding, the assignment (SETQ) operates as a pattern matching process with possible establishment of decision points. In addition, there are several types of expressions (statements with which the user can explicitly build such points, e.g., GOAL, ATTEMPT, etc.).

In the discrimination net, each expression exists only once. Similar to atoms in LISP, expressions representing factual statements can be assigned properties. These can be manipulated directly with the functions PUT, GET, and ERASE. Moreover, one can make general queries about the existence of expressions of a certain form (described by a pattern) with specific properties.

Similar to variable binding environments in LISP, a certain state of the discrimination net with its expressions and their properties can be considered a binding. The QA4 term for this is context. A new context level is built whenever a function or PROG is entered. Unlike LISP, the user can refer to any context in both value assignments (SETQ) and PUT operations on P-lists. Finally, the user can create contexts.

A control structure revived by QA4 is the demon. This is a function that is only activated when certain changes are made in the database (in the property structures of certain expressions in the discrimination net). In QA4, a WHEN statement is formulated for this purpose:

(WHEN Condition THEN Action).

The action corresponds to a function call. In the condition, the changes (access with GET – keyword SENDS, assignment with PUT – keyword RECEIVES, etc.), as well as the location where these changes occur (value or property), and any restrictions on the values of the changes can be specified.

It would go too far to discuss all the new ideas that flowed into QA4. It is certain that QA4 represents a milestone in coping with programming problems in the field of Artificial Intelligence.

In real-world applications, QA4 was not as groundbreaking as its conception may suggest. This is a typical problem in the development of LISP metasystems: the techniques to build such a system are quite easy to handle. However, processing within the metasystem runs somewhat slowly, as there is usually a double interpretation. This leads to efficiency problems. Often, after a short period of use, so many new ideas emerge that the old system is forgotten without regret. The cause of this approach is the sometimes rapid progress of artificial intelligence science.

7.7 MLISP 2

In the years 1971-72, the metasystems at SRI and Stanford University reached their final version. They are both characterized by being considered metasystems until the end.

At Stanford University, MLISP was frequently used, prompting the implementors, Enea and Smith, to consider further development. They emphasized the compiler-compiler line, taking into account new ideas developed with PLANNER and QA4.

The use of MLISP suggested that it would be desirable to design the syntax of MLISP in such a way that it can be easily modified. This would make it possible to accept programs in various Lisp metasystems that appear externally similar. The experience of translating a convenient programming language into the internally useful and effective Lisp structure led to the goal of creating a metasystem that could treat any other programming languages in the same way. The syntax analysis method of top-down analysis, inherent in Lisp, was enhanced with backtracking to expand the class of processable languages.

Inspired by PLANNER, the authors had the opportunity to explore current implementations of backtracking. They arrived at an independent and efficient solution.

Smith and Enea completed the work on MLISP 2 around 1971-1972. In the main information source [SM73a], they reported on a system that had been operational for over two years.

MLISP 2 is primarily referred to in [SM73a] as a base system for writing compilers. It is characterized by the following features: the notation of MLISP, language extensibility (which applied not only to MLISP 2 but also allowed defining new languages), pattern matching, and backtracking.

MLISP 2 is considered, in many respects, upward compatible with MLISP; the main changes do not modify the syntax of the language but rather the control structure or the execution system of MLISP, significantly expanding its

applications. Some minor changes [SM73a][S.12], such as eliminating BEGIN-END brackets, allowing any number of qualifiers after a base expression (i.e., repeated consideration of the result of a function application as a function and argument of the same) [SM73a][S.182], a convenient notation for P-list processing – the dot notation [SM73a][S.20], and the inadmissibility of vector assignments [SM73a][S.18], are intended to improve the language's readability.

The "major" changes compared to MLISP are embodied in the LET and SELECT expressions.

The LET expression combines a syntax pattern matcher and a semantic expression evaluator. It can be used to extend the language, define entirely new languages, or serve as a pattern matcher in any context. In a sense, it is the heart of the MLISP 2 extension mechanism [SM73a][S.24].

However, the view of the applicability of pattern matching and its inclusion in MLISP 2 is a restriction compared to the possibilities in PLANNER, for example. In MLISP 2, the pattern matcher is only intended for processing linear input data (strings) since it is mainly used as a tool for compiler construction.

The patterns specified in a LET expression can be formulated in a notation that can rightfully be called a syntax description language. This language consists of four basic constructions (literals, non-terminals, inline expressions, and meta-expressions) and certain additional elements. The authors believed that their approach could describe any context-free or context-sensitive grammar during the processing of a language while being able to change the grammar [SM73a].

The processing mechanism based on this description language gains its power from the pattern matcher and the built-in backtracking capabilities. All three meta-expressions[31] initiate the establishment of decision points during pattern matching, which can be reverted to in case of errors [SM73a][S.36ff].

Characteristic of language extension or description using the LET expression is the fact that the syntax rules specified as patterns are immediately translated and compiled into LISP programs, while the semantic routines remain expressed in LISP expressions [SM73a][S.26,77].

The backtracking implemented in MLISP 2 is mainly reflected in the SELECT expression, ignoring the implicit elements within the pattern matcher.

SELECT represents a generalization of Floyd's choice function CHOICE [SM73a][S.50], [SM73b]. The SELECT expression requires five specifications for executing partial actions, which, if omitted, are replaced by default information. These specifications are: a value function, a range from which to select, a successor function (determining the next element), a termination condition, and finally, a termination function. These functions are represented by expressions that essentially represent the LAMBDA bodies [SM73a][S.50]. If the termination condition is met, the termination function is applied to the value range. It may happen that the FAILURE function explicitly triggers backtracking, leading back to the decision point established by SELECT [SM73a][S.49ff].

MLISP 2 provides the user with pure data backtracking [SM73a], where

[31] REP - repetition, ALT - alternative, OPT - optional element

returning to a decision point leads to a restoration of variable values at that point [SM73b]. The implementation follows a very efficient strategy, and for the user, the basic philosophy is simple, plausible, and transparent.

The outstanding features of MLISP 2 led to the implementation of a whole series of well-known and important language processing systems, among which Milner's implementation of D. Scott's logic for computable functions has been the most significant [MIL72a]. Additionally, a deduction system [WEY74], a parser for English, an English-French translator, an ALGOL compiler, and MLISP 2 itself are mentioned [SM73b][S.680]. Not to forget is also the Pascal-LISP converter by R.L. London and D.C. Luckham [IG73].

The next step was supposed to be the metasystem LISP 70. By perfecting the available control structures (backtrack and coroutine control structure) and expanding data manipulation capabilities (pattern matching, streaming, long-term memory for large databases, data typing, pattern-directed computation), they hoped to achieve a new quality of a base system for artificial intelligence work.

According to reports [SM73a][S.2], [TES73], a first version of LISP 70 ran in June 1973, but a real (production) version was never fully implemented.

What is interesting about LISP 70 is that its authors, in a sense, were once again thinking about developing a successor to LISP. In doing so, they revived many features of the unsuccessful LISP 2: the union of LISP 1.5 and ALGOL in syntax and semantics [TES73][S.671], compiling all functions, etc.

The basic philosophy of LISP 70 starts with an extensible language kernel that the user can expand by defining new tools and by incorporating more efficient implementation elements for often-used parts of the language. For this adaptability towards extension and specialization, the authors used a machine-independent low-level language (code for an ideal LISP machine [TES73][S.671]) to ensure the transferability of the language.

Thus, the LISP 70 compiler generates programs in this ML language, which must then be translated into object code. Only the final part is machine-dependent. The compiler is part of the EVAL function—each argument of EVAL is compiled concerning the current environment and then executed as machine code. Parts of the symbol tables and the entire compiler are constantly present in the system.

Efficiency requirements are supposed to be fulfilled by a particularly fast pattern-based translator whose rules are compiled into short and efficient code.

Like LISP 2, the "LISP for the 70s" disappeared quite quickly from the stage. The reasons for this seem to lie in the authors' change from Stanford University to other workplaces.

MLISP 2 was only used for a short time for convenient compiler writing. Due to the dual interpretation (MLISP 2 programs are translated into LISP and interpreted by a special processing program, which, in turn, is executed in the LISP system), excessive processing times were recorded. Although the reports [SM73a, TES73] are almost known among LISP users everywhere, the system was never exported. Thus, the user base remained very small and local.

```
   LET PATTERN (L) =  REP 1 M *  OPT '! OPT '# ALT [IDENTIFIER] | '[TOKEN]
| '< [IDENTIFIER] !'> | '[ !'] |
   MEAN L;
```

Fig. 17. An MLISP 2 example program [SM73a][S.78]

7.8 From PLANNER to CONNIVER

During the implementation and application of MICROPLANNER, the behavior of the programmer and their problems could be studied. It quickly became clear that there were still many things to be desired, especially concerning efficiency.

Even Hewitt admitted that the PLANNER procedures were genuinely inefficient [HEW71b][p.21]. Sussman and McDermott, as implementors, concluded that automatic backtracking was primarily to blame because it led to poor programming practices [SUS72a][p.7]. Their main thesis is that essential parts of the control structures, backtracking, and multiprocessing only served the bad programmer - the good one could do without them [HEW71b][p.13].

Based on their study of PLANNER and the use of MICROPLANNER, McDermott and Sussman developed their own language, which they called CONNIVER [SUS72b, MCD72].

Although Sussman and McDermott's criticism primarily targets backtracking, it is easy to see that they are actually opposed to the fundamental concept of PLANNER. Not coincidentally, they express this in the title of their manifesto: "Conniving is better than Planning!" (Not caring about anything (?) is better than foresight.) The core of their analysis is undoubtedly the statement: "What is needed is a programming language, not a theorem prover" [SUS72a][p.30].

The two authors want to force the programmer to think carefully about their programs and not burden a poorly processed problem onto a too general mechanism without analysis. However, they are also simultaneously against the principle of not treating a problem by developing a self-created program but describing it and letting the system find the solution using this description. In this regard, Hewitt is also correct when he quotes Hayes [HEW75a], who observes that a step backward has been taken ("... something of a retrograde step ... [HEW75a][p.190]).

Nevertheless, CONNIVER represents an interesting development. Like MICROPLANNER, CONNIVER is presented as an interpreted language, and the processing system was written in LISP.

CONNIVER, based on an implementation model by Bobrow and Wegbreit [BO73c], uses, in addition to the tree of control information (control environments), a database tree called the context tree. The user only deals with the last branch of this tree: each change affects only the data in the currently active branch. Users can easily switch contexts to transfer local information to an outer context.

Each user can build their control structures, forcing them to think more about the flow. Unlike PLANNER, the procedural entities corresponding to PLANNER's theorems, called methods here, do not work with the first (or next)

element from the database during pattern matching. Instead, they are provided with a list of possibilities that they must manage themselves. CONNIVER provides language structures in which this work can be noted.

Due to the explicit management of the program flow, CONNIVER users can devise and implement any kind of flow planning.

The pattern matcher works with similar syntactic structures as MICROPLANNER but is capable of processing arbitrarily structured patterns.

All parts of the syntax foreign to LISP in both MICROPLANNER and CONNIVER are implemented using MACRO characters in MACLISP. The respective character activates a LISP program, which may prompt for further input and, as a result, provides the desired internal structure. This capability has led to the fact that MACLISP mostly implemented metasystems that are largely LISP-like in their syntax.

CONNIVER retains from PLANNER the pattern-directed procedure call. During pattern matching, the possibility lists consider not only elements from the database that directly correspond to the pattern (items) but also the methods that would lead to such a matching element. If the mechanism for trying out possibilities (TRY-NEXT) finds such an option, the procedure associated with it must be activated.

The dispute over CONNIVER and PLANNER has enlivened many meetings of specialists in the field of Artificial Intelligence since 1972. There may have been some heated debates at MIT as early as 1971.

```
(IF-NEEDED WINMODES (WINMODE? PLAYER? SQUARE? MOVE) "AUX" (PLAYER
SQUARE MOVE (CONTEXT (PUSH 'CONTEXT)) P1 SQ1 P2 SQ2) (ADD '(HAS ,PLAYER
,SQUARE)) (REMOVE '(FREE ,SQUARE)) (CSETQ P1 (FETCH '(HAS ,PLAYER ?SQ1)))
:OUTERLOOP (TRY-NEXT P1 '(GO 'END)) (CSETQ P2 (FETCH '(HAS ,PLAYER
?SQ2))) :INNERLOOP (TRY-NEXT P2 '(GO 'OUTERLOOP)) (COND ((AND (LESSP
SQ1 SQ2) (CSETQ MOVE (THIRD-IN-ROW SQ1 SQ2)) (PRESENT '(FREE ,MOVE)))
(NOTE (INSTANCE))) ) (GO 'INNERLOOP) :END (ADIEU) ).
```

Fig. 18. A CONNIVER example program [SUS72a][p.26]

In Hewitt's contribution to the 2nd IJCAI 1971, these controversies and the work between 1970 and 1971 are practically not addressed in a single word. Meanwhile, the ideas associated with PLANNER were directed towards two main complexes: The problem of representing knowledge about the "real world" in the form of procedures (procedural knowledge representation) - as opposed to storing facts using data - plays a central role alongside the pattern-directed multiprocess backtrack control structure and has proven to be a key idea for the further development of the science of Artificial Intelligence.

The work [HEW71a], in its condensed form, provides a good overview of the state of development in the fundamental ideas of PLANNER.

According to Hewitt [HEW71a][p.168], the thesis of procedural embedding of knowledge means that intellectual structures should be analyzed through their procedural analogies. This can be interpreted, for example, in the following way: Descriptions are procedures that assess how well certain candidates satisfy these

descriptions; patterns are descriptions comparable to data configurations. Data types are patterns used in declarations of the allowed ranges of procedures and variables. In general, data types have their analogies in the way procedures form, destroy, observe, and transform data...

"Models of programs are procedures for defining properties of procedures and for verifying these properties..." [HEW71a][p.168].

The thesis is beautifully illustrated as follows: "The procedural analogy of a drawing is a procedure that executes the drawing" [HEW71a].

In the control flow, special consideration should be given to backtracking. Here, specifically, the return to earlier system states is meant. Thus, an attempt can be made that uses the available knowledge, and if it does not succeed, a choice is possible that can also use the knowledge collected during the unsuccessful attempt. The specific implementation of backtracking in PLANNER reflects the fact that all actions can be gradually undone: It is just as easy to work through a program forwards as it is backward.

The concept of multiprocessing (where multiple places in the program are active simultaneously - almost more of a futuristic concept in 1973) is intended to improve control flow. Several pattern search processes through a problem space are assumed to be executed simultaneously.

By focusing on working with patterns, PLANNER combines aspects of working with data structures and controlling this work through control structures. Patterns are used in the construction and decomposition of structured data. The search for data similar to a given pattern initially occurs directly in stored data in the database. If, in the use of an initially matching datum, it is found that it does not correspond to the further purpose, an error is triggered, and it returns to the choice situation. If the database is exhausted (i.e., searched), procedures are sought that generate similar data. Therefore, procedures must not only do their work but must also be examined: Each procedure should have a pattern associated with what the procedure aims to produce [HEW71c][p.172].

With the further progress of the PLANNER concept, Hewitt increasingly moved away from LISP. In his doctoral thesis, which he presented in 1972, he described a system that was only conditionally implementable within LISP [HEW71c]. The pattern matching language MATCHLESS is presented in a significantly expanded form. Not only is the spectrum of variable prefixes - with value use (.), value use if value exists or assignment (?) and assignment (local with _, global with :) - extended and doubled (! denotes segment orientation before the second prefix) [HEW71c][p.62], but also the declaration scheme for variable types [HEW71c][p.66].

Patterns are in general lists that include constants, variables, sub-patterns (i.e., sublists), function calls (syntactically indicated with angle brackets, they provide elements; indicated with curly brackets, they provide segments), vectors (bounded by square brackets), and actor expressions. Function calls with a number in a functional position serve as access to vector elements. Actor expressions do not provide values like function calls, i.e., constants, but patterns.

The concept of Actors would gain significantly greater importance for PLANNER systems in the further development in 1971.

Over time, Hewitt has tried to incorporate all reasonably progressive ideas in programming methodology and Artificial Intelligence into the PLANNER formalism. Thus, the complex around PLANNER becomes a system of ideas about the fundamentals of Artificial Intelligence, general ideas about structuring programs, and the required expressive power of programming languages. In doing so, Hewitt has occasionally revealed a certain eclecticism.

An example of this is his work at the 3rd IJCAI 1973 in Stanford [HEW73]. Presented is a loose collection of ideas for a new formalism: PLANNER73. Although quite relevant concepts are represented, this work is extremely difficult to read and is more of a mixture of an advertising campaign for the Actor formalism, Lewis-Carroll quotes, and fragmentary elements of the formal language, further enriched with the polemic against Sussman and McDermott, which becomes even more intense in later publications.

Within the scope of this book, we cannot delve into the problems associated with PLANNER73 and PLASMA. Although a substantial part of LISP is still visible behind the formalism, PLANNER73 should no longer be referred to as an LISP metasystem (not to mention the lack of implementation). Despite the often peculiar form of the associated publications, the PLANNER line can be regarded as an influential and promising approach. Further literature: [PROM11, LAB12, LAB13, LAB14].

7.9 QLISP, CLISP, CGOL, and RLISP

When INTERLISP became available at SRI and one could study the way Teitelman had integrated the CLISP subsystem into the main LISP system, a similar transformation process began for QA4. By mid-1973, an interim result was achieved, and by the end of 1975, the final result was QLISP.

Similar to QA4, all data structures in QLISP are stored in a permanent database called the discrimination network. Upon admission to the network, expressions are converted into canonical form, so that each possible expression has a single representation and, therefore, a unique position in the network.

The spectrum of data structures in QLISP is expanded to include vectors. While vectors structurally correspond to tuples, i.e., an ordered sequence of elements, they are evaluated differently. The value of a vector is the vector of the elements. The SET data structure in QLISP is named CLASS.

Among other changes ([REB73], p. 2), noteworthy are the extensions made to functions for insertion and search in the discrimination network. Their syntax has been standardized so that each of these expressions (statements, as in QA4) can refer to a specific context. Additionally, each statement can cite a set of daemon functions (referred to as a team of functions) and can itself be labeled with a name that a function from the team can reference.

Daemon functions were introduced by QA4. These are functions that are never explicitly called but are activated upon the occurrence of certain conditions. These conditions are particularly related to the addition of new data to the database. One can imagine the implementation such that every change in the database triggers a search through all activation conditions of the daemons.

All daemons affected by the new situation in the database are implicitly called within the change action.

```
(HITCH [QLAMBDA (HAPPY ← HUMAN) (BIS (PERSON ← Y) SEX [@ (PARTNERSEX('
(PERSON $HUMAN] THEN (ASSERT(MARRIED $HUMAN $Y) APPLY $MARRIAGEDEMONS)
(QPUT (PERSON $HUMAN) MARRIEDTO $Y WRT GLOBAL APPLY $COMPUTERELATIONS)
('(HAPPY $HUMAN])
```

Fig. 19. A QLISP example program ([REB73], p. 29)

In addition to QLISP, the following metasystems are used today: In STANFORD LISP 1.6, mainly MLISP, as well as in UNIVAC LISP. In STANFORD LISP/360, it is RLISP, the ALGOL-like formulation language of the Form Manipulation System REDUCE, used worldwide. In various dialects of MACLISP, CGOL is used, and in INTERLISP, CLISP takes its place.

CLISP is embedded in the comprehensive INTERLISP system through error handling. CLISP programs are not transferred into the internal LISP via a preprocessor, as in MLISP, but remain as they are: In ALGOL-like form, with IF-THEN-ELSE statements, FOR-DO loops, etc. However, during the initial evaluation, the CLISP expression is replaced by the equivalent LISP expression. The interpreter cannot process these expressions, reports errors, and initiates translation as error handling. Because of this approach, CLISP can be mixed freely with LISP. All correct LISP expressions are processed immediately - the CLISP handler never sees them.

This type of embedding has the advantage that there is no double interpretation. Moreover, error handling programs, especially the correction program, work fully for CLISP.

The LISP expressions derived from CLISP expressions are directly inserted into the program at the location of the CLISP prototype for simpler constructions (infix, prefix, and IF-THEN-ELSE notations). In other cases, the CLISP handler uses a hash list, from which an entry point is derived from the CLISP expression. The translation result can be found there. This last variant applies to many forms of cyclic expressions but also to pattern matches and record expressions (i.e., more general data structures).

The pattern description language is as flexible as in MLISP 2. It can describe not only elements but also segments. Additionally, as in all more advanced pattern description languages, comparison conditions and partial assignments are possible.

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)) FOR OLD X FROM M TO N DO
(PRINT X) WHILE (PRIMEP X) (FOR I FROM 1 TO 5 SUM 1 2) (BIND X WHILE
X  (READ)  = 'STOP DO (PRINT(EVAL X))) (FOR X IN Y AS I FROM I TO 10
COLLECT X) [PRODUCT; ($$VAL← $$VAL*BODY); (FIRST $$VAL← 1] (FIND
X IN Y SUCHTHAT X MEMBER Z) X: ($2 Y← $3) X : (FOO← $ 'A - )  (REVERSE
FOO) ('A ← FOO 'D $)
```

Fig. 20. CLISP program lines ([TE74], §23)

The record processing system is intended to allow the user a certain data structure-independent programming style ([TE74-], p. 23, 50ff.). All in all, CLISP [TE76] is likely to be significantly superior to MLISP. This is not only due to the richer capabilities of the language itself but also to its better integration into the host system. Another, unmentioned indication is the translation aid included in the CLISP processing system.

CGOL [PRAT76] is similar to MLISP. It was developed around 1971 by V. Pratt as an external representation of LISP. He continued the work around 1973, and by 1975, the system was available. The syntax was designed to be optimally processed according to Pratt's operator precedence method. This makes language processing much faster, and the syntax also includes certain dynamic elements. Like CLISP, CGOL also has a reverse translator. In MACLISP, it is connected to the LISP system in such a way that the output can fundamentally be in CGOL.

Thus, CLISP and CGOL can be seen as external languages over the LISP system.

```
   define „GC-DAEMON" (spacelist); let total_accessible = 0.0, total_consed
= 0.0; for element in spacelist do (let space = car(element), freebefore
= cadr(element), freeafter = cadd r( element), sizebfore = caddd r(
element), sizeafter = car(cddddr(element)); consed ofq space : = sizebefore-freebefore-acce
ble ofq space; total_consed :== total_consed + consed ofq space; accesslble
ofq space := sizeafter-freeafter; total_accessible := total_accessible
+ acceslble ofq space);
```

Fig. 21. Part of a CGOL program ([BAK77b], p. 4)

```
   SYMBOLIC PROCEDURE APPLY(FN, X, A); IF ATOM FN THEN IF FN EQ 'CAR
THEN CAAR X ELSE IF FN EQ 'CDR THEN CDAR X ELSE IF FN EQ 'CONS THEN
CAR X. CADR X ELSE IF FN EQ 'ATOM THEN ATOM CAR X ELSE IF FN EQ 'EQ
THEN CAR X EQ CADR X ELSE APPL Y(EVAL(FN, A), X, A) ELSE IF CAR FN
EQ 'LAMBDA THEN EVAL(CADDR FN, PAIRLIS(CADRFN,X,A)) ELSE IF CAR FN
EQ 'LABEL THEN APPLY(CADDR FN, X, (CADR FN, CADR FN), A);
```

Fig. 22. An RLISP example program ([LUX75a], p. 20)