

The Evolution of Lisp

Guy L. Steele Jr.
Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142
Phone: (617) 234-2860
FAX: (617) 243-4444
E-mail: gls@think.com

Richard P. Gabriel
Lucid, Inc.
707 Laurel Street
Menlo Park, California 94025
Phone: (415) 329-8400
FAX: (415) 329-8480
E-mail: rpg@lucid.com

Abstract

Lisp is the world's greatest programming language—or so its proponents think. The structure of Lisp makes it easy to extend the language or even to implement entirely new dialects without starting from scratch. Overall, the evolution of Lisp has been guided more by institutional rivalry, one-upsmanship, and the glee born of technical cleverness that is characteristic of the “hacker culture” than by sober assessments of technical requirements. Nevertheless this process has eventually produced both an industrial-strength programming language, messy but powerful, and a technically pure dialect, small but powerful, that is suitable for use by programming-language theoreticians.

We pick up where McCarthy's paper in the first HOPL conference left off. We trace the development chronologically from the era of the PDP-6, through the heyday of Interlisp and MacLisp, past the ascension and decline of special purpose Lisp machines, to the present era of standardization activities. We then examine the technical evolution of a few representative language features, including both some notable successes and some notable failures, that illuminate design issues that distinguish Lisp from other programming languages. We also discuss the use of Lisp as a laboratory for designing other programming languages. We conclude with some reflections on the forces that have driven the evolution of Lisp.

Introduction	2
Implementation Projects Chronology	5
From Lisp 1.5 to PDP-6 Lisp: 1960–1965	6
MacLisp	7
Interlisp	15
The Early 1970's	21
The Demise of the PDP-10	22
Lisp Machines	25
IBM Lisps: Lisp360 and Lisp370	30
Scheme: 1975–1980	32
Prelude to Common Lisp: 1980–1984	36
Early Common Lisp	36
Other Stock Hardware Dialects: 1980–1984	44
Standards Development: 1984–1990	49
Evolution of Some Specific Language Features	60
The Treatment of NIL (and T)	60
Iteration	63
Macros	66

Numerical Facilities	78
Some Notable Failures	81
Lisp as a Language Laboratory	88
Why Lisp is Diverse	96

1 Introduction

1.1 Organization of This Paper

A great deal has happened to Lisp over the last thirty years. We have found it impossible to treat everything of interest coherently in a single linear pass through the subject, chronologically or otherwise. Projects and dialects emerge, split, join, and die in complicated ways; the careers of individual people are woven through these connections in ways that are sometimes parallel but more often orthogonal. Ideas leap from project to project, from person to person. We have chosen to present a series of slices through the subject matter. This organization inevitably leads to some redundancy in the presentation.

Section 2 discusses the history of Lisp in terms of projects and people, from where McCarthy left off [McCarthy, 1981] up through the efforts to produce official standards for Lisp dialects within IEEE, ANSI, and ISO. Section 3 examines a number of technical themes and traces separately their chronological evolution; here the emphasis is on the flow of ideas for each topic. Section 4 traces the use of Lisp as a language laboratory and implementation tool, especially for the development of AI languages; particular attention is paid to ways in which feedback from the AI languages influenced the development of Lisp itself. Section 5 draws some conclusions about why Lisp evolved as it did.

1.2 The Pattern of Language Evolution

The evolution of Lisp since Lisp 1.5 is characterized by a cycle of diversification, acceptance, and consolidation. During diversification, new language constructs, new styles of programming, new implementation strategies, and new programming paradigms are experimented with and introduced to existing Lisp dialects, or new Lisp dialects are designed. In either case, a new Lisp dialect is, in effect, created. During acceptance, these new Lisp dialects are either accepted or rejected. The designers or backers of the new dialect will set conditions for acceptance, and the success of the acceptance phase will be determined by those conditions. During consolidation, a variety of dialects are merged into a mainstream dialect. Typically one dialect will be chosen as the root and the branches will be taken from the same tree as the root or from other dialects. Consolidation is a process of standardization, either formal or informal. Consolidation, when most successful, results in a stable development platform.

The cycle is not inexorable: The process can break down, stop and start, and cycles can be skipped. But it is possible to view the process of evolution at any point as being within one of these stages of the cycle, with the goal of moving to the next stage.

Each stage can be characterized by the conditions that allow it to be entered.

1.2.1 Conditions for Acceptance

The most critical stage is acceptance. This stage determines which language features and paradigms will be part of the next stage of stable development. The conditions for acceptance are being on the right machines, fitting into local user models, solving a pressing new problem, having the right *cachet*. Each will be explored in detail.

The acceptance stage depends on a particular acceptance group choosing to base their work on the new dialect. An example acceptance group comprises the commercial artificial intelligence companies. A particular dialect of Lisp might be targeted to solve the problems of this community through, for example, integration with mainstream languages. Whether this dialect moves on to the next stage in the cycle—consolidation—depends on whether this acceptance group chooses to enter a period of acceptance with the dialect and in fact the dialect passes the acceptance stage by actually providing solutions.

The dialect should run on the right machines. This includes being on the right manufacturers' machines. A Lisp dialect will be accepted when the people who will determine acceptability can use the dialect right away with proper performance. This also includes having acceptable size and performance for the machines. If the key user groups are using computers of a certain size and speed, the Lisp dialect should run acceptably on that configuration. This is complicated by the fact that the designers or promoters of a dialect will sometimes choose the initial target computer with an eye only towards the total number of installed computers rather than the more restrictive number of computers installed or soon-to-be installed in the target user base. As we will see, InterLisp and NIL fell into disuse when their promoters chose the wrong computers.

Next is fitting into local user models. Each user group will have a style of working or set of methodologies within which the new dialect must fit. Oftentimes a new dialect will be an extension of an existing one, and unless the acceptance group is able to embrace the dialect in a timely fashion, the dialect will not enter the acceptance stage and will unlikely be part of a consolidation. Sometimes the acceptance group is newly formed and has no existing working styles or methodologies, and in this case the new dialect should present a working model acceptable to the acceptance group.

Next is solving timely problems. An acceptance group is one whose success will determine the acceptance of a dialect if that dialect is required by the acceptance group. The acceptance group must have a set of needs not addressed at the time the new dialect would enter this stage; otherwise the acceptance group would have no need to switch to the new dialect, and perhaps such a switch would be too risky for the group. If the group is commercial or has a set of outside-determined success criteria, that group will be risk averse, and the new dialect must offer significant perceived value for achieving these criteria. For example, a group that is embarking in a large project with strict productivity or reuse requirements, a Lisp dialect with a strong object-oriented component might be readily embraced. A dialect with marginal or incremental improvements will often not be accepted.

Finally is *cachet*. We use the word 'cachet' as it is used in the advertising and fashion industries. A scarf with cachet is worn by people in the most exclusive and desired echelons of high society, and so that scarf is highly desired and sought after. A perfume with cachet is charged, prized, and people don't merely want to own it—they want to possess it.

So it is with computers. The best example—though some might argue with it—is the Macintosh as compared with the IBM PC. When you look at them objectively, there is considerably more software on the PC, and it is probably used much more heavily in business than the Macintosh. But there is something about the Macintosh that appeals to the leading edge computerist. If you look at most hobbyists—the ones who rave on and on about how great computers are and how everyone should have one—they either own a Macintosh or wish they did. The difference between a PC and a Macintosh is cachet.

Cachet does not always mean best of breed or most expensive or most exclusive. A 1967 Ford Mustang has more cachet than an 1992 Mercedes 300 SE, though the latter is far more expensive.

Acceptance groups comprise people who must individually wish to adopt a new dialect. The people who would use Lisp generally characterize themselves as leading edge. Therefore, there

must be at least some aspects of leading edge technology in the new dialect. For example, CLOS lends cachet to Common Lisp—almost causing Common Lisp with CLOS to be regarded as a new dialect—because it is perceived as an advance over mainstream object-oriented programming.

1.2.2 Conditions for Consolidation

Consolidation is the stage during which standards—formal or informal—are set. Sometimes several dialects are merged through a combination of a standardization process and a design rationalization process. When this happens one dialect will usually be chosen as the base and features from other accepted dialects will be blended into it.

The conditions for entering consolidation are that the dialect or dialects have been accepted, that the acceptance group is in ascendancy, and that the dialect is perceived to have helped the acceptance group and will continue to help.

First, the dialect must be accepted. This implies not only that the dialect met the functional needs of the group—its features were useful—but that the dialect fit well into the group and met its performance and size goals.

Next, the acceptance group must be poised for growth. This is an important external factor for the success of a dialect of Lisp, and possibly for other languages. The user group that would use the language must itself be successful or be perceived to about to be successful. Thus, a language faces a double hurdle: To succeed with a particular group and to have appealed to a group that will itself succeed. When coupled with the conditions for entering the acceptance stage, this implies that the language must also have succeeded in choosing the right target platform.

Furthermore, when an acceptance group is growing or about to grow, that group will typically be growing in size and in geographical extent, and so exchanging software becomes important. This happens both when the acceptance group is commercial and when it is research-oriented.

Finally is the perceived contribution of the dialect to the success or success conditions of the acceptance group. That is, the acceptance group must either be successful or be perceived to about to be successful, and the dialect must appear to have contributed to that situation. It is usually not sufficient that the acceptance group succeed or that the dialect have been merely useful to the effort that put the acceptance group where it is: The dialect must be perceived as a necessary part of the effort. Otherwise the acceptance group will be tempted to reevaluate the language decision, and in the case of Lisp, once this reevaluation is entered upon, it is difficult for Lisp to be retained.

1.2.3 Conditions for Diversification

Diversification happens when old solutions are inadequate or when there is nothing else to do. The conditions for entering diversification are that external driving factors are languishing, that there is a retreat to smaller research or development groups, and that the previous dialects have failed in some way.

First, when the acceptance group is in descendency, there are fewer resources to allocate for language implementation improvement and incremental design. When a language has been through the consolidation stage, it is subject to incremental improvement through implementation improvements and design improvements: performance or size can be improved, and some new, minor language features can be added, but paradigm shifts require a new cycle of diversification, acceptance, and consolidation.

The language experts will no longer be obligated to make these small improvements and will instead turn their attention to solving such problems as those which caused the acceptance group to decline. Perhaps a new acceptance group will be targeted along with its problems. Or perhaps

another language or languages will have such cachet or success that that language will be mined for incorporation into a new dialect.

Next, such retreat creates smaller groups, pockets of language groups. Because innovation typically involves small groups, this is virtually a necessary condition for new design. The real importance is that the language design groups—though they might not be called such—will be free from interruption to pursue their new designs.

Finally, the previous solutions must have had failures for the acceptance group. If the acceptance group declines because of extraneous economic factors, for example, there would be no need to pursue change. Though this has never stopped those intent on change for change's sake. Some call this the I-Did-It-My-Way syndrome.

Diversification comes from many sources. When a language fails, there may be attempts to fix the problems by retreating to earlier principles and redesigning; there may be another language with a different paradigm that appears suitable for the solution to problems the dialect failed to solve, or perhaps the cachet of that language will be irresistible; or pure intellectual or scientific curiosity will lead a designer down a path that results in new language features or paradigms.

1.2.4 Pollenation

Language design and evolution is driven by people, whose careers carry them from one set of concerns to another. And like a snowball rolled along a thin layer of snow, people pick up influences from the problems they work on. And so we observe that particular individuals enter and leave our story of the diversification of Lisp, and when they reappear after an absence, they have new experiences under their belts and will apply those experiences. New players will appear and interact as people with others, and the results of those interactions—the languages—will reflect intellectual affinities to a wide variety of other languages, language concepts and features, and language paradigms.

Acceptance groups are people, and the whole story of language evolution and diversification is against a background of human concerns and institutions. Despite the fact that appeal is made to objective criteria for language design, the inevitable humanness shines through.

2 Implementation Projects Chronology

The following sections describe the events in the evolution of Lisp from 1960 until the present. Each era or situation is first described as a pure chronology and then the events of the era or situation are placed within the context of the cycle of diversification-acceptance-consolidation, and the important technical concepts are described.

Early thoughts about a language that eventually became Lisp started in 1956 when John McCarthy attended the Dartmouth Summer Research Project on Artificial Intelligence. Actual implementation began in the fall of 1958. In 1978 McCarthy related the early history of the language [McCarthy, 1981], taking it approximately to just after Lisp 1.5. See also [McCarthy, 1980]. We start with a brief description of the implementations of Lisp starting with Lisp 1.5 and ending with MacLisp and Interlisp. At that point we will delve into MacLisp and Interlisp more deeply, tracing their development and the differing philosophies they embraced. Other Lisp dialects of the same period will be presented next, including Scheme, Portable Standard Lisp, Zetalisp, and the other precursors to Common Lisp. Common Lisp will be described in detail, including its standardization and implementations. The blending of object-oriented techniques into Lisp will be examined, especially with an eye towards the Common Lisp Object System (CLOS). European and Japanese

Lisp evolution will be examined, but in less detail, except for EuLisp, which is a dialect of Lisp defined in opposition to Common Lisp.

2.1 From Lisp 1.5 to PDP-6 Lisp: 1960–1965

During this period, Lisp spread rapidly to a variety of computers, either by bootstrapping from an existing Lisp on another computer or by a new implementation. In almost all cases, the Lisp dialect was small and simple; the implementation straightforward. There were very few changes made to the original language.

In the early 1960's, Timothy Hart and Thomas Evans implemented Lisp 1.5 on the Univac M 460, a military version of the Univac 490. It was bootstrapped from Lisp 1.5 on the IBM 7090 using a cross-compiler and a small amount of machine language code for the lowest levels of the Lisp implementation [Hart, 1985].

Robert Saunders and his colleagues at System Development Corporation implemented Lisp 1.5 on the IBM-built AN/FSQ-32/V computer—the Q-32 [Saunders, 1985b]. The implementation was bootstrapped from the IBM 7090 and PDP-1 computers at Stanford University.

The PDP-1 Lisp at Stanford was implemented by John McCarthy and Steve Russell.

In 1963, L. Peter Deutsch (at that time a high school student) implemented a Lisp similar to Lisp 1.5 on the PDP-1 at Bolt Beranek and Newman (BBN) [Deutsch, 1985]. This Lisp was called Basic PDP-1 Lisp.

By 1964 a version of Lisp 1.5 was running in the Electrical Engineering Department at MIT on an IBM 7094 computer, running the Compatible Time Sharing System (CTSS). [?]. This Lisp and Basic PDP-1 Lisp were the main influences on the PDP-6 Lisp [PDP-6 Lisp, 1967] implemented by DEC and some members of MIT's Tech Model Railroad Club in the spring of 1964. This Lisp was the first program written on the PDP-6. Also, this Lisp was the ancestor of MacLisp, the Lisp written to run under the Incompatible Time Sharing System (ITS) [Eastlake, 1968; Eastlake, 1972] at MIT on the PDP-6 and later on the PDP-10.

At BBN a successor to Basic PDP-1 Lisp was implemented on the PDP-1 and an upward-compatible version, patterned after Lisp 1.5 on the MIT CTSS system, was implemented on the Scientific Data Systems 940 (SDS 940) by Daniel Bobrow and D. L. Murphy. A further upward-compatible version was written for the PDP-10 by Alice Hartley and Murphy, and this Lisp was called BBN Lisp [Teitelman, 1971]. In 1973, not long after the time that SDS was acquired by Xerox and renamed Xerox Data Systems, the maintenance of BBN Lisp was shared by BBN and Xerox Palo Alto Research Center and the name of the Lisp was changed to Interlisp [Teitelman, 1974].

The PDP-6 [DEC, 1964] and PDP-10 [DEC, 1969] computers were, by design, especially suited for Lisp, with 36-bit words and 18-bit addresses. This allowed a `CONS` cell—a pair of pointers or addresses—to be stored efficiently in a single word. There were half-word instructions that made manipulating the `CAR` and `CDR` of `CONS` cells very fast. The PDP-6 and PDP-10 also had fast, powerful stack instructions, which enabled fast function calling for Lisp.

Almost all of these implementations had a small hand-coded (assembly) core and a compiler; the rest of the Lisp was written in Lisp and compiled.

In 1965, virtually all of the Lisps in existence were identical or differed only in trivial ways. After 1965—or more precisely, after MacLisp and BBN Lisp diverged from each other—there came a plethora of Lisp dialects.

Analysis

These early Lisp dialects fit into the pattern typical of the diversification stage:

During this period there was little funding for language work, the groups were isolated from each other, and each group was directed primarily towards serving the needs of the local acceptance group, which was limited to a handful of researchers. The typical situation is characterized by the description “an AI lab with a Lisp wizard down the hall”. During this period there was a good deal of experimentation with implementation strategies. There was little thought of consolidation, particularly in the form of a formal standards process, partly because of the pioneering feeling that each lab embodied.

The first real standard Lisps were MacLisp and Interlisp, and as such they deserve some attention.

2.2 MacLisp

MacLisp was the primary Lisp dialect at the MIT AI Lab from the late 1960’s until the early 1980’s. Other important Lisp work at the Lab during this period included Lisp-Machine Lisp (later named Zetalisp) and Scheme. MacLisp is usually identified with the PDP-10 computer, but MacLisp also ran on another machine, the Honeywell 6180, under the Multics operating system [Organick, 1972].

2.2.1 Early MacLisp

The distinguishing feature of the MacLisp/Interlisp era is the attention to production quality or near production quality implementations. This period saw a consolidation of implementation techniques, with great attention to detail.

For example, in Lisp 1.5 a special variable is a binding of an identifier to a location, a pointer to which is placed on the property list of the symbol whose name is the identifier. When a programmer declares `x` to be special, the compiler and loader conspire to create a symbol named `x`. The property list for `x` has a property with the indicator `special`, and its value is a cell. That cell is the location to which `x` refers and where assignments to `x` place values. When `x` is let-bound (the name `x` is newly bound), the old value stored in this cell is placed on the control stack along with enough information to restore the old value to this cell when control returns past the LET-binding point.

A common variable is one in which the value cell is used to hold the value, and binding a common variable requires the use of an explicit a-list. Evaluating a common variable at run time requires a call to EVAL.

Special variables can be shared among different compiled functions, but cannot be shared with interpreted functions; common variables can be shared by compiled functions and interpreted functions, but are relatively much slower than special variables.

In terms of implementation, special variables are the precursors of what is now known as shallow binding, while common variables are the precursors of deep binding.

In shallow binding, the value of a special variable is always kept in the value cell of a symbol. let-binding the symbol causes the old value to be placed on the control stack and the new value to be placed in the value cell. References to the value of the symbol within the dynamic scope of the LET will see this new value. When control exits the LET, the old value is restored.

In deep binding, the identifier-value pair is placed on the control stack when let-binding occurs. Reference to the value of a symbol first looks at the stack for any bindings there, and if none are found, the value cell is referenced. Searching the stack can be done relatively efficiently by linking together special-binding blocks (the places where identifier-value pairs are stored on the stack), by putting a flag in the symbol stating whether the symbol has been bound (and therefore should

be looked-up on the stack), or by caching special variables in stack frames in which they will be frequently referenced. These techniques can be used alone or in combinations.

In the early MIT PDP-6 Lisp [PDP-6 Lisp, 1967], Richard Greenblatt decided that having both common and special variables, as in Lisp 1.5, was inelegant; he removed **COMMON** variables from the language, but made **SPECIAL** variables work using the value cell. This was the first implementation of Lisp to use what is called *shallow binding*.

A key difference between MacLisp and Interlisp was the approach to syntax. MacLisp favored the pure list style, using **EVAL** as the top level. Interlisp, along with Lisp 1.5, used **EVALQUOTE**. To create a dotted pair of the atoms **a** and **b** in MacLisp, one would type this expression to **EVAL**:

```
(cons (quote a) (quote b))
```

or, using the syntactic abbreviation `'x` for `(quote x)`,

```
(cons 'a 'b)
```

In Lisp 1.5, one could type an expression (actually two expressions) like this to **EVALQUOTE**:

```
cons(a b)
```

The “quote” in the name **EVALQUOTE** signifies the “implicit quoting of the arguments” to the function applied. MacLisp forked off and used **EVAL** exclusively as a top level interface, while BBN-Lisp (and thus Interlisp) accommodated both, using **EVAL** if the input was one form and **APPLY** if the input line was two or more forms.

The phrase “quoting arguments” actually is misleading and imprecise. It refers to the actions of a hypothetical preprocessor that transforms the input from a form like `cons(a b)` to one like `(cons 'a 'b)`. The term **EVALQUOTE** thus arose through an incorrect understanding of the original model. A similar misunderstanding carried over into the description of the so-called **FEXPR** or “special form”. In some texts on Lisp one will find descriptions of special forms that speak of a special form “quoting its arguments” when in fact a special form has a special rule for determining its meaning and that rule involves not evaluating some forms [Pitman, 1980].

McCarthy [McCarthy, 1981] noted that the original Lisp interpreter was regarded as a universal Turing machine: It could perform any computation given a set of instructions (a function) and the initial input on its tape (arguments). Thus it was intended that `cons(a b)` be regarded not as a mutated version of `(cons (quote a) (quote b))`, but as a function and (separately) a literal list of arguments. In hindsight we see that the **EVALQUOTE** top level might better have been called the **APPLY** top level, making it pleasantly symmetrical to the **EVAL** top level; the BBN-Lisp documentation brought out this symmetry explicitly. (Indeed, **EVALQUOTE** would have been identical to the function **APPLY** in Lisp 1.5 if not for these two differences: (a) in Lisp 1.5, **APPLY** took a third argument, an environment (regarded nowadays as something of a mistake that resulted in dynamic binding rather than the lexical scoping needed for a faithful reflection of the lambda calculus); and (b) “**EVALQUOTE** is capable of handling special forms as a sort of exception” [McCarthy, 1962]. Nowadays such an exception is referred to as a *kluge* [Raymond, 1991]. (Note that MacLisp’s **APPLY** function supported this kluge.)

A special form is a Lisp expression whose meaning is determined by a special rule rather than by the default rule of evaluating all subforms to produce a function and a sequence of arguments on which to invoke it. For example, the meaning or operation of the expression `(+ a b)` is determined by evaluating **a** and **b**, and applying the function `+` to the results. The meaning or operation of the expression `(if a b c)` is determined by evaluating the expression **a**, and if the value is not **NIL**,

the expression `b` is evaluated as the value of the `if`, and if so the expression `c` is evaluated as its value. In this sense `if` is called a special form.

Actually, the term special form in reference to `if` is misleading because `if` is actually the name of the operator used to distinguish a form that is to be taken as a special form. Some refer to `if` as a special operator.

This is an example of a trend we will see several times in the Lisp community—the misuse or confusing use of terminology.

In Lisp 1.5 and MacLisp special forms are implemented by placing a function on the property list of the symbol whose print name corresponds to the special form's name. For example, in MacLisp, `COND` has an `FSUBR` property, where the “f” in `FSUBR` signifies a special form, and the “subr” signifies a compiled subroutine. The evaluation process for arguments is then left up to the programmer. Here is how `IF` could be defined as a special form in MacLisp (an `FEXPR` is the interpreted version of an `FSUBR`):

```
(defun IF fexpr (form)
  (let ((predicate (car form))
        (then (cadr form))
        (else (caddr form)))
    (cond ((eval predicate) (eval then))
          (t (eval else))))))
```

This code is not as efficient as it could be, but it illustrates that special forms can be extended by user-written code.

MacLisp introduced the `LEXPR`, which is a type of function that takes any number of arguments and puts them on the stack; the single parameter of the function is bound to the number of arguments passed. The form of the lambda-list for this argument—a symbol and not a list—signals the `LEXPR` case. Here is an example of how to define `LIST`, a function of a variable number of arguments that returns the list of those arguments:

```
(defun LIST n
  (do ((i n (1- i))
        (answer () (cons (arg i) answer)))
      ((zerop i) answer)))
```

Parameter `n` is bound to the number of arguments passed. The expression `(arg i)` refers to the i^{th} argument passed.

The need for the `LEXPR` (and its compiled counterpart, the `LSUBR`) arose from a desire to have variable arity functions such as `+`. Though there is no semantic need for an n -ary `+`, it is convenient for programmers to be able to write `(+ a b c d)` rather than the equivalent but more cumbersome `(+ a (+ b (+ c d)))`. Later we will see at least one other place where the notion of a `LEXPR` will be useful for implementing new language features during diversification.

The simple but powerful macro facility on which `DEFMACRO` is based was introduced in MacLisp in the mid-1960's. See section 3.3.

Other major improvements over Lisp 1.5 were arrays; the modification of simple predicates—such as `MEMBER`—to be functions that return useful values; `PROG2`; and the introduction of the function `ERR`, which allowed user code to signal an error.

In Lisp 1.5, certain built-in functions might signal errors, when given incorrect arguments, for example. Signaling an error normally resulted in program termination or invocation of a debugger. Lisp 1.5 also had the function `ERRSET`, which was useful for controlled execution of code that *might* cause an error. The special form

(`errset form`)

evaluates *form* in a context in which errors do not terminate the program or enter the debugger. If *form* does not cause an error, `ERRSET` returns a singleton list of the value. If execution of *form* does cause an error, the `ERRSET` form quietly returns `NIL`.

MacLisp added the function `ERR`, which signals an error. If `ERR` is invoked within the dynamic context of an `ERRSET` form, then the argument to `ERR` is returned as the value of the `ERRSET` form.

Programmers soon began to use `ERRSET` and `ERR` not to trap and signal errors but for more general control purposes (dynamic non-local exits). Unfortunately, this use of `ERRSET` also quietly trapped unexpected errors, making programs harder to debug. A new pair of primitives, `CATCH` and `THROW`, was introduced into MacLisp in June 1972 so that `ERRSET` could be reserved for its intended use of error trapping.

Analysis

The lesson of `ERRSET` and `CATCH` is important. The designers of `ERRSET` and `ERR` had in mind a particular situation and defined a pair of primitives to address that situation. However, the construction of these primitives is in two parts: one part that traps and ignores errors, and another part that transferred control to a dynamically earlier point. Because there were no such non-local control transfer primitives, programmers began to use the existing facilities in unintended ways. Then the designers had to go back and split off the desired functionality. The pattern of design (careful or otherwise), unintended use, and later redesign is common.

MacLisp was written as a large assembly language core, an interpreter, and a compiler. Above we saw that the developers of MacLisp were both consolidating some of the ideas from other languages, typically by rationalization, and were diversifying the language with new data structures.

The next phase of MacLisp development began when the developers of MacLisp started to see a large and influential acceptance group emerge—Project MAC and the Mathlab/Macsyma group. The emphasis turned to satisfying the needs of their user community rather than doing language design and implementation as such.

2.2.2 Later MacLisp

During the latter part of its lifecycle, MacLisp adopted language features from other Lisp dialects and from other languages, and some novel things were invented.

The most significant development for MacLisp occurred in the early 1970's when the techniques in the “fast arithmetic compiler” `LISCOM` [Golden, 1970] were incorporated into the MacLisp compiler. This new compiler, `NCOMPLR` [Moon, April 1974; White, 1969–1982; Pitman, 1983], would become a standard against which all other Lisp compilers were measured in terms of the speed of running code. Inspired by the needs of the MIT Artificial Intelligence Laboratory (AI Lab), whose needs covered the numeric computations done in vision and robotics, several new ways of representing and compiling numeric code resulted in numeric performance of compiled MacLisp on a near par with `FORTRAN` compilers [Fateman, 1973].

`LISCOM` was largely the work of Jeff Golden. The relative performances of compiled numeric Lisp code and `FORTRAN` numeric code on the PDP-10 stirred the Digital compiler writers to improve the DEC `FORTRAN` compiler to the point where it was difficult for the Lisp compilers to compete. The relative performance of subsequent Lisp compilers on numeric code has never reached the high-water mark achieved in 1972, but some Lisp compilers have done very well.

Bignums—arbitrary precision integer arithmetic—were added in 1970 or 1971 to meet the needs of Macsyma users. The code was a more or less faithful transcription of the algorithms in [Knuth,

1969]. Later Bill Gosper suggested some improvements, notably a version of **GCD** that combined the good features of the binary GCD algorithm with Lehmer’s method for speeding up integer bignum division [Knuth, 1981, ex. 4.5.2-34].

In 1973 and 1974, David Moon led an effort to implement MacLisp on the Honeywell 6180 under Multics. As a part of this project he wrote the first truly comprehensive reference manual for MacLisp, which became familiarly known as the “Moonual” [Moon, April 1974].

Richard Greenblatt started the MIT Lisp Machine project in 1974; David Moon, Richard Stallman, and many other MIT AI Lab Lisp hackers eventually joined this project. As this project progressed, language features were selectively retrofitted into PDP-10 MacLisp as the two projects cross-fertilized.

Complex lambda lists partly arose by influence from MDL [Galley, 1975], which was a language for the Dynamic Modeling Group at MIT. It ran on a PDP-10 located in the same machine room as the AI and Matlab machines. The austere syntax of Lisp 1.5 was not quite powerful enough to express clearly the different roles of arguments to a function. Complex lambda-lists appeared as a solution to this problem and became widely accepted; eventually they terribly complicated the otherwise elegant Common Lisp Object System.

MDL—its earlier, less-palatable-to-funders name was “Muddle”—was a Lisp-related language, also developed at MIT, that had a strong influence on mainstream Lisp evolution. It introduced lambda-list options such as **"OPTIONAL"** to denote optional arguments, **"AUX"** for defining auxiliary variables, **"ARGS"** for a **&rest** variable (one that is bound to a list of the remaining arguments), and some others that never made it into mainstream Lisp dialects. MDL had *locatives*, which might be described as “lvals” (in the C sense), only first-class; essentially they are pointers to memory locations and can be treated like variables to read and write those locations (in C++ this concept is known as *references*). MDL had a splicing marker to enable a sequence of values (called a segment) to be spread as arguments; the splicing marker **,@** used in Common Lisp’s backquote syntax is descended from this. MDL also introduced through example the use of a final **?** in a name to connote a predicate, in contrast to mainstream Lisp dialects, which used a trailing **P**. (Scheme now uses the “sensible” **?** convention and Common Lisp the “traditional” **P** convention, so the predicate that is true of numbers and only numbers is called **NUMBER?** in Scheme and **NUMBERP** in Common Lisp.)

Lisp-Machine Lisp would pick up the lambda lists and the locatives, and would adapt the evaluation and splicing markers in backquote, though the evolution of the concept passes through Conniver [**?**] where it is refined into a form very similar in appearance and effect as in Lisp-Machine Lisp, but in fact the implementation is quite different. Scheme would pick up the use of a trailing **“?”** for predicates. MDL had a multitasking facility that would reappear in various forms in Lisp-Machine Lisp, Conniver, and various implementations of Common Lisp.

MacLisp introduced the notion of *read tables*. A read table provides programmable input syntax for programs and data. When a character is input, the table is consulted to determine the syntactic characteristics of the character for use in putting together tokens. For example, the table is used to determine which characters denote whitespace. In addition, functions can be associated with characters, so that a function is invoked whenever a given character is read; the function can read further input before returning a value to be incorporated into the data structure being read. In this way the built-in parser can be reprogrammed by the user. This powerful facility made it easy to experiment with alternative input syntaxes for Lisp, ranging from such simple abbreviations as **'x** for (**quote x**) to the backquote facility and elaborate Algol-style parsers. See section 3.7.1 for further discussion of some of these experiments.

With the read table facility, several different surface syntaxes for MacLisp have been used, for example CGOL [Pratt, 1976].

A continuing problem with Lisp has been the hesitancy of people to adopt it because its syntax is unfamiliar and forbidding to some. Many programmers prefer an Algol-like syntax. In 1991, the CAD Framework Initiative group (??? reference is to a recent Scheme Digest message) voted to use Scheme as its extension language, but expressed a desire to use a C-like surface syntax. Similarly, the 1989 Common Prototyping Language study group [?] recommended using a language with many Lisp features but with a familiar surface syntax.

In 1971, Jon L White (sic) changed the representation of the data structure that maps names (strings) to symbols from a list to a hash table [White, 1969–1982]; thus the traditional `OBLIST` (list of objects, that is, named atoms) of Lisp 1.5 was renamed the `OBARRAY`. This signaled a move to a more careful choice of data structures to optimize size or speed. Later implementations of Lisp relied even more heavily on hash tables for parts of their implementations. The introduction of the `OBARRAY` marked an important step in the increase of sophistication of Lisp implementors.

Jon L White was commonly known as “Jonl”, which can be pronounced as either “jónnell” like “O’Donnell” or “john-ell”. Because of this, his name is usually written “Jon L White” rather than the correct “John L. White.”

Big improvements to the MacLisp I/O system were implemented in 1974. Prior to that point, a MacLisp program could have only one input file and one out file open at a time, plus separate connections to the keyboard and terminal screen. Under the so-called `NEWIO` system, the number of open files was limited only by the operating system. (ITS afforded an upper limit of 16, which in that day was considered a very large number!) Under `NEWIO` files, or rather open file connections, became first-class objects that could be passed as arguments, returned as values, and garbage-collected. (One of the trickiest parts of the implementation was ensuring that files were closed cleanly as file objects were reclaimed.)

In the days before Unix became widespread, there was as much experimentation in the design of operating systems as in the design of programming languages. Just as each laboratory might have its own dialect of Lisp, it was not unusual for each laboratory to run a different operating system—if not completely home-grown, then a local mutation of one obtained elsewhere. In 1978, there was a desire to share MIT software written in MacLisp with other institutions. While most major laboratories also had a PDP-10, if not two, none of them ran ITS. MacLisp therefore had to be ported to five other PDP-10 operating systems:

- TOPS-10, DEC’s own operating system for the PDP-10, in which users and file directories were identified by two 18-bit integers expressed in decimal, such as `483,11823`
- CMU TOPS-10, which had been locally modified at Carnegie-Mellon University to identify users and file directories by mnemonic identifiers such as `N920JW51` rather than pairs of integers (in this example, `JW` stands for Jon White, `5` means the account was assigned in 1978, and `1` means he was the second user with initials `JW` that year—note that two letters and two digits are encodable in 18 bits)
- WAITS [?], a much more extensively mutated version of TOPS-10 at Stanford, which identified users and file directories by two words of up to three characters each (such as `LSP,DOC` for Lisp documentation or `1,RPG` for Richard P. Gabriel—note that three 6-bit characters (no lower case) fit into 18 bits!)
- TENEX, developed at BBN and radically different from TOPS-10, with user, directory, and file names of up to 31 characters
- TOPS-20, the result of DEC’s taking TENEX to heart and gussying it up

This porting effort was straightforward but tedious, requiring hundreds of conditional-assembly commands to be inserted into the master assembly-language source files for MacLisp, mostly in the I/O routines.

Analysis

MacLisp can be seen as one consolidation of the flurry of Lisp implementations in the early 1960's. There was a particular acceptance group—Project MAC—that drove consolidation into a stable, high performance implementation of a derivative of Lisp 1.5. Therefore, we can see the first example of the cycle: diversification during the Lisp 1.5 era, acceptance at the start of the MIT Project MAC era, consolidation during the heyday of Project MAC, and, as we'll see, a decline of funding for Lisp at MIT preceding a period of diversification.

During the period from 1969 until 1981, MacLisp enjoyed several acceptance groups: from 1969 until around 1973 it was the AI Lab, in particular the vision group. From at 1972 until around 1981 it was the Mathlab/Macsyma group, though the earlier Mathlab group under William Martin was a strong influence.

The AI Lab conducted research into artificial intelligence generally, but it focussed on vision, robotics, natural language, planning, and representation. In addition there was some interest in language design for AI, as exemplified by Carl Hewitt, Terry Winograd, Gerry Sussman, and Guy Steele.

The Mathlab and Macsyma groups were interested in symbolic mathematics, which is a discipline that develops algorithms, data structures, and programs to symbolically manipulate the structures and concepts of mathematics. For example, symbolic differentiation (as opposed to numeric differentiation) is one of the first symbolic mathematics programs.

From 1972 to 1983 ??? the support for MacLisp was provided by the Macsyma group which had Department of Energy funding for the Macsyma Consortium and some new development. The Macsyma Consortium was a group of institutions, for example Lawrence Livermore National Laboratories ???, that used Macsyma for their work. Generally these members were also funded by DOE.

Thus MacLisp had an acceptance group, which had accepted MacLisp as its standard. Nevertheless, because MacLisp ran only on PDP-10's, there was little need to standardize the language through consolidation with other languages. However,

MacLisp adopted only a small number of features from other Lisp dialects. In 1974, about a dozen persons attended a meeting at MIT between the MacLisp and Interlisp implementors, including Warren Teitelman, Alice Hartley, Jon L White, Jeff Golden, and Guy Steele. There was some hope of finding substantial common ground, but the meeting actually served to illustrate the great chasm separating the two groups, in everything from implementation details to overall design philosophy. (Much of the unwillingness of each side to depart from its chosen strategy probably stemmed from the already severe resource constraints on the PDP-10, a one-megabyte, one-MIPS machine. With the advent of the MIT Lisp Machines, with their greater speed and *much* greater address space, the crowd that had once advocated a small, powerful execution environment with separate programming tools embraced the strategy of writing programming tools in Lisp and turning the Lisp environment into a complete programming environment.) In the end only a trivial exchange of features resulted from “the great MacLisp/Interlisp summit”: MacLisp adopted from Interlisp the behavior (`CAR NIL`) $\rightarrow$ `NIL` and (`CDR NIL`) $\rightarrow$ `NIL`, and Interlisp adopted the concept of a read table.

The adoption of the Interlisp treatment of `NIL` was not received with universal warmth. We quote the public announcement by Jon L White [White, 1969–1982, July 9, 1974, item 2]:

For compatibility with Interlisp (`foo`), the `CAR` and `CDR` of `NIL` are always but always `NIL`. `NIL` still has a property list, and `GET` and `PUTPROP` still work on it, but `NIL`'s property list is not its `CDR` (crock, crock). The claim is that one can write code such as (`CADDR X`)

instead of the more time- and space-consuming

```
(AND (CDR X) (CDDR X) (CADDR X))
```

and so on. Send complaints to GLS, but I doubt it will do you any good.

A few words of explanation are in order. Part of the source of the problem is that, in MacLisp, CDR applied to an atomic symbol returned the symbol's property list. With this change, the symbol NIL becomes unlike all other symbols in there being no way to get its property list. That CDR happened to work this way on symbols was an accident of implementation that users began to rely on heavily—this is another example of the unintended use of a feature that we saw with ERRSET and ERR. The contract for CDR, then, was not uniform: when applied to dotted pairs it did one thing, and when applied to symbols it did an abstractly unrelated thing that happened to be implemented by the same machine instruction. The solution was to complete the set of abstract operations for property lists; PLIST and SETPLIST were introduced to MacLisp in 1975, about nine months after the Interlisp compatibility change, so that CDR and RPLACD need not be used on symbols.

These were the sources of information about MacLisp that users could reference:

1. The original PDP-6 Lisp manual [PDP-6 Lisp, 1967] and an update [?].)
2. LISP ARCHIV [White, 1969–1982], a complete on-line log listing each newly added and deleted feature of the language. With each new release of MacLisp, release notes were prepended to the log. This file was maintained from 1969 until 1981, when Jon L White left MIT for Xerox. (The strange spelling is a consequence of the design of ITS, which limited file names to two components of at most 6 character each. On TOPS-10, with its 6-and-3 limits and different punctuation, the file was called LISP.ARC.)
3. The 1974 edition of the MacLisp Manual [Moon, April 1974], written by David A. Moon as part of the effort to put MacLisp up on Multics. This valuable document was usually referred to orally as the “Moonual.”
4. Jon L White published a short history of MacLisp [?].
5. *The Revised MacLisp Manual* by Kent Pitman, written in 1983.
6. “The wizard down the hall”: implementors Jon L White and Guy Steele at MIT, and local MacLisp gurus such as Richard P. Gabriel at Stanford University, Rodney A. Brooks at ??? University in Australia, David Touretzky at CMU, and Timothy Finin at the University of Illinois. Like the exiles in Ray Bradbury's *Fahrenheit 451*, these perhaps fanatical hackers *were* the documentation and would recite necessary information on request—a deplorable situation, perhaps, from today's perspective, but workable in a small community working with a rapidly changing piece of software.
7. The source code. Part of the spirit of the MacLisp community was the knowledge that when there was any doubt about what the language did, one could read the code.

Note that MacLisp ran primarily on PDP-10's, and there was a single, central set of source code files, so its relatively small and tightly-knit community felt little need for a separate, complete language specification. (The Moonual, produced when MacLisp began to straddle two machine architectures, is the exception that proves the rule; but the rule stood, for the PDP-10 enclave regarded the Multics community as outsiders, welcome to try to keep up as new changes were announced in LISP ARCHIV. The Moonual was not revised for another nine years, when Kent Pitman produced what became known as the “Pitmanual” [Pitman, 1983].

The Interlisp effort, by contrast, produced a comprehensive reference manual for its more far-flung user community and a separate virtual machine specification that aided in porting Interlisp to different computer architectures. As the MacLisp community metamorphosed into the MIT Lisp

Machine community, better documentation became a necessity and eventually appeared, thanks again to David Moon, this time with Daniel Weinreb [Weinreb, November 1978]. (This one was called the “chine nual” (pronounced *sheenua*) because the title **Lisp Machine Manual** in big block letters was wrapped around the entire paperback cover so that only those letters showed on the front.)

Several other programming languages were developed within MacLisp: Micro-Planner, Coniver, and Scheme to name a few. Several object-oriented extensions to MacLisp were tried out, including Extend and Flavors. Extend [Burke, 1983] is a mechanism to extend the builtin Lisp type system using inheritance, and Flavors will be discussed in detail below.

By the mid-1970’s it was becoming increasingly apparent that the address space limitation of the PDP-10—256k 36-bit words, or about one megabyte—was becoming a severe constraint as the size of Lisp programs grew. MacLisp by this time had enjoyed nearly 10 years of strong use and acceptance within its somewhat small but very influential user community. Its implementation strategy of a large assembly language core would prove to be too much to stay with the dialect as it stood, and intellectual pressures from other dialects, other languages, and the language design aspirations of its implementors would result in new directions for Lisp.

To many, the period of stable MacLisp use was a golden era in which all was right with the world of Lisp. (This same period is also regarded today by many nostalgics as the golden era of Artificial Intelligence.) By 1980 the acceptance group for MacLisp was on the decline—the funding for Macsyma would not last too long. Various funding crises in AI had depleted the ranks of the AI Lab Lisp wizards, and the core group of wizards from MIT and MIT hangers-on moved to new institutions. The late part of the 1970’s and early part of the 1980’s was a period of diversification.

Several names we have seen will reappear—White, Steele, Moon, Greenblatt, McCarthy, Deutsch, Bobrow, and Gabriel—and the rest will disappear.

2.3 Interlisp

Interlisp (and BBN-Lisp before it) introduced many radical ideas into Lisp programming style and methodology. The most visible of these ideas are embodied in programming tools, such as the spelling corrector, the file package, DWIM, CLISP, the structure editor, and MASTERSCOPE.

The origin of these ideas can be found in Warren Teitelman’s PhD dissertation on man-computer symbiosis [Teitelman, 1966]. In particular, it contains the roots of structure editing (as opposed to “text” or “tape” editing [Rudloe, 1962]), breakpointing, advice, and CLISP. (William Henneman in 1964 described a translator for the A-language [Henneman, 1985], an English-like or Algol-like surface syntax for Lisp (see section 3.7.1), but it was not nearly as elaborate or as flexible as CLISP. Henneman’s work does not appear to have directly influenced Teitelman; at least, Teitelman does not cite it, though he cites other papers in the same collection containing Henneman’s paper [Berkeley, 1985].)

The spelling corrector and DWIM were designed to compensate for human foibles. When a symbol had no value (or no function definition), the Interlisp spelling corrector [Teitelman, 1974] was invoked, because the symbol might have been misspelled. The spelling corrector compared a possibly misspelled symbol with a list of known words. The user had options for controlling the behavior of the system with respect to spelling correction. The system would do one of three things: (a) correct automatically; (b) pause and ask whether a proposed correction were acceptable; or (c) simply signal an error.

The spelling corrector was under the general control of a much larger program, called DWIM, for “Do What I Mean”. Whenever an error of any sort was detected by the Interlisp system, DWIM was invoked to determine the appropriate action. DWIM was able to correct some forms

of parenthesis errors, which, along with the misspelling of identifiers, comprised the most common typographical errors by users.

DWIM would not have been especially useful if correcting errors were not done permanently. (Who wants to acknowledge the same correction day after day?) Interlisp did not maintain a file of function definitions in the same way as, say, MacLisp. To a MacLisp user, a file of ASCII characters was the primary representation of a program; these files were then read into a running system or precompiled into binary code for later loading. In Interlisp, files were also used as a nonvolatile storage medium, external to the running Lisp system, for storing user source code; but the user seldom examined his file directly, nor cared about its format. All modifications to user code were done within the Interlisp programming environment. An Interlisp user did not modify files using an ASCII editor, but instead used a resident Lisp structure editor to edit the S-expression representation of the program.

The ideal situation that was approximated by Interlisp, using files, was that the user interacted with a Lisp system that never terminated, and it was the program as it was in use that was important, not its representation as text in a file. When errors were corrected by DWIM, the in-memory program representation was altered to reflect the correction and, as an almost incidental side effect from the user's point of view, the file that served as the nonvolatile backup medium for the program representation also was updated. Thus, once a bug was corrected, it was permanently corrected.

CLISP (Conversational LISP) was a mixed Algol-like and English-like syntax embedded within normal Interlisp syntax. Here is a valid definition of FACTORIAL written in Interlisp CLISP syntax:

```
DEFINEQ((FACTORIAL
  (LAMBDA (N) (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)))))
```

CLISP also depended on the generic DWIM mechanism. Note that it not only must, in effect, rearrange tokens and insert parentheses, but also must split atoms such as `N=0` and `N*` into several appropriate tokens. Thus the user need not put spaces around infix operators.

CLISP defined a useful set of iteration constructs. Here is a simple program to print all the prime numbers p in the range $m \leq p \leq n$:

```
(FOR P FROM M TO N DO (PRINT P) WHILE (PRIMEP P))
```

CLISP, DWIM, and the spelling corrector could work together to recognize the following as a valid definition of FACTORIAL [Teitelman, 1973]:

```
DEFINEQ((FACTORIAL
  (LAMBDA (N) (IFFN=0 THENN 1 ESLE N*8FACTORIALNN-1)))))
```

Interlisp eventually “corrects” this mangled definition into the valid form shown previously. Note that shift-8 is left parenthesis on the Model 33 teletype, which had a bit-paired keyboard. DWIM had to be changed when typewriter-paired keyboards (on which left parenthesis was shift-9, and shift-8 was the asterisk) became common.

(Critics of Interlisp, particularly those who were proponents of MacLisp, often commented, rightly or wrongly, that DWIM seemed to be tuned to the particular typing mistakes to which Teitelman was prone, and no others. These critics were given to lampooning DWIM with such expansions as “Do What Teitelman Means” or “Do What Interlisp Means.” “Let the system automatically ‘correct’ programs? I take a very DWIM view of that, sir!” MacLisp proponents would remark that MacLisp used DDWIDM: “Don’t Do What I Didn’t Mean.”)

Interlisp had a structure editor. Lisp code was displayed and altered within the Interlisp system. Operations in the editor applied to the current S-expression—as selected by the user—or to a small

surrounding context of the current expression. This editor was developed when teletypes and other slow, printing terminals were standard; the Interlisp structure editor enables a programmer to edit Lisp source code very efficiently on such terminals. This style of editing is natural to people who are programming almost exclusively in Lisp, though some people prefer always to edit ASCII text.

MASTERSCOPE was a facility for finding out information about the functions in a large system. MASTERSCOPE could analyze a body of code, build up a data base, and answer questions interactively. MASTERSCOPE kept track of such relationships as which functions called which others (directly or indirectly), which variables were bound where, which functions destructively altered certain data structures, and so on. (MacLisp had a corresponding utility called INDEX, but it was not nearly as general or flexible, and it ran only in batch mode, producing a file containing a completely cross-indexed report.)

Interlisp introduced the concept of *block compilation*, in which multiple functions are compiled as a single block; this resulted in faster function calling than would otherwise have been possible in Interlisp.

Interlisp ran on PDP-10's, Vaxen (plural of Vax [Raymond, 1991]), and a variety of special-purpose Lisp machines developed by Xerox and BBN. The most commonly available Interlisp machines were the Dolphin, the Dorado, and the Dandelion (collectively known as D-machines). The Dorado was the fastest of the three, and the Dandelion the most commonly used. It is interesting that different Interlisp implementations used different techniques for handling special variables: Interlisp-10 (for the PDP-10) used shallow binding, while Interlisp-D (for D-machines) used deep binding. These two implementation techniques exhibit different performance profiles—a program with a certain run time under one regime could take 10 times longer under the other.

This situation of unexpected performance is prevalent with Lisp. One can argue that programmers produce efficient code in a language only when they understand the implementation. With C, the implementation is straightforward because C operations are in close correspondence to the machine operations on a Von Neumann architecture computer. With Lisp, the implementation is not straightforward, but depends on a complex set of implementation techniques and choices. A programmer would need to be familiar not only with the techniques selected, but the performance ramifications of using those techniques. It is little wonder that good Lisp programmers are harder to find than good C programmers.

Just as MacLisp's I/O system was designed around the facilities provided by ITS, Interlisp's I/O system was designed around those of TENEX, which, all things considered, were in many ways more advanced than those of ITS (though more awkward in such areas as interrupt handling) and offered a friendlier user interface. Files in Interlisp were represented everywhere by name; every I/O function (read and write, not just open and close) accepted a character string naming the file. Thus one could have many files open simultaneously, but files were not represented by first-class Lisp objects subject to automatic reclamation.

Like MacLisp, Interlisp extended the function calling mechanisms in Lisp 1.5 with respect to how arguments can be passed to a function. Interlisp function definitions specified arguments as the cross product of two attributes: LAMBDA versus NLAMBDA, and spread versus nospread.

LAMBDA functions evaluate each of their arguments; NLAMBDA functions evaluate none of their arguments (that is, the unevaluated argument *subforms* of the call are passed as the arguments). Spread functions require a fixed number of arguments; nospread functions accept a variable number. These two attributes were not quite orthogonal, because the parameter of a nospread NLAMBDA was bound to a list of the unevaluated argument forms, whereas the parameter of a nospread LAMBDA was bound to the number of arguments passed and the ARG function was used to retrieve actual argument values. There was thus a close correspondence between the mechanisms of Interlisp and MacLisp:

<u>Interlisp</u>	<u>MacLisp</u>
LAMBDA spread	EXPR
LAMBDA nospread	LEXPR
NLAMBDA spread	no equivalent
NLAMBDA nospread	FEXPR

There was another important difference here between MacLisp and Interlisp, however. In MacLisp, “fixed number of arguments” had a quite rigid meaning; a function accepting three arguments must be called with exactly three arguments, neither more nor less. In Interlisp, any function could legitimately be called with any number of arguments; excess argument forms were evaluated and their values discarded, and missing argument values were defaulted to NIL. This was one of the principal irreconcilable differences separating the two sides at their 1974 summit. Thereafter MacLisp partisans derided Interlisp as undisciplined and error-prone, while Interlisp fans thought MacLisp awkward and inflexible, for they had the convenience of optional arguments, which did not come to MacLisp until `&optional` and other complex lambda-list syntax was retrofitted, late in the game, from Lisp-Machine Lisp.

One of the most innovative of the language extensions introduced by Interlisp was the *spaghetti stack* [Bobrow, 1973]. The problem of retention (by closures) of the dynamic function-definition environment in the presence of special variables was never completely solved until spaghetti stacks were invented.

The idea behind spaghetti stacks is to generalize the structure of stacks to be more like a tree, with various branches of the tree subject to retention whenever a pointer to that branch is retained. That is, parts of the stack are subject to the same garbage collection policies as are other Lisp objects. Unlike closures, the retained environment captures both the control environment and the binding environment.

Spaghetti stacks, *per se*, are an efficient implementation of tree-structured stacks using a linear stack; in the situation where parts of the stack are not retained, there is almost no performance penalty incurred.

Interlisp retained the Lisp 1.5 flavor to a greater extent than did MacLisp. MacLisp changed the order of some arguments to some functions (such as MAPCAR and the other MAP functions) while Interlisp retained the original order. Interlisp programming style is heavily influenced by the programming environment, CLISP, and DWIM; therefore the Lisp portion of Interlisp remains very similar to early Lisps, the effort in improving programming style focusing on CLISP and DWIM. In dialects like MacLisp, in which a “Lispy” style was retained, the Lisp part of the language itself was advanced.

One of the minor, but interesting, syntactic extensions that Interlisp made was the introduction of the superparenthesis, or superbracket. If a right square bracket] is encountered during a read operation, it balances all outstanding open left parentheses, or back to the last outstanding left square bracket [. Here is a simple example of this syntax:

```
DEFINEQ((FACTORIAL
  (LAMBDA (N)
    (COND [(ZEROP N) 1]
          (T (TIMES N (FACTORIAL (SUB1 N))
```

Analysis

MacLisp and Interlisp came into existence about the same time and lasted about as long as each other. They differed in their acceptance groups, though any generic description of the two groups

would not distinguish them: both groups were researchers at AI labs funded primarily by ARPA (later DARPA), and these researchers were educated by MIT, CMU, and Stanford. The principal implementations ran on the same machines, and one had cachet as the Lisp with the nice environment while the other was the lean, mean, high-powered Lisp. The primary differences came from different philosophical approaches to the problem of programming. There were also different pressures from their user groups; MacLisp users, particularly the Mathlab group, were willing to use a less integrated programming environment in exchange for a good optimizing compiler and having a large fraction of the PDP-10 address space left free for their own use. Interlisp users preferred to concentrate on the task of coding by using a full, integrated development environment.

The MacLisp philosophy centered on a high quality implementation, performance, text editors, and language expressiveness. The Interlisp philosophy centers on the programming environment and the task of coding—back then they would call it the task of programming.

MacLisp was a file-based, text-editor-oriented environment, while Interlisp was residential with an in-memory (in-core) Lisp structure editor. The MacLisp environment used external tools linked loosely together, while the Interlisp environment was tightly coupled with the language implementation—in fact, the environment and the language shared the same address space, and actually were part of the same program.

The Interlisp philosophy was developed largely by Teitelman in the middle 1960's, at which time there were very few large screen terminals—the printing teletype-type terminal was most common. And the primary text editors were “tape editors.” A tape editor is a text editor that operates on the text as if it were a long linear tape. Commands moved backwards and forwards on the tape, either by certain distances or by searching. Text could be inserted and deleted. One popular editor at the time, TECO, was originally called the Tape Editor and Corrector (later Text Editor and Corrector). In short, the process of entering and correcting code was very difficult outside the Interlisp environment, and so the goal of the Interlisp development environment was to simplify and shorten that process—even typing code was difficult and so shortcuts were sought.

Teitelman's approaches were new and exciting, and certainly they improved coding productivity.

The alternative, MacLisp, had certain advantages as well. Because the language implementation was file-based and text-based, the development environment was captured by independent external tools, which could follow their own evolutionary process, and MacLisp programmers could simply inherit the benefits. Until Emacs [?] appeared, though, there were few benefits to be had.

When the tools are separate, the user does not need to accept a set of tools along with the language. With a package to choose, and risk that the environmental tools might be unsuitable, the cost of choosing a language plus environment might be too high for some.

However, MacLisp programmers wished to have their productivity improved—actually, they probably didn't care about productivity, they simply wanted their job simplified and made more fun. The result was that MacLisp developers focussed more on the language itself, adding macros and new data structures, and improving performance.

Another set of differences in philosophy lies in pretty-printing. The following is an example of the definition of the function, `MEMBER`, in Lisp 1.5. It is reproduced exactly as it appeared in the *Lisp 1.5 Programmer's Manual* [McCarthy, 1962]:

```
DEFINE((
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
                             ((EQ A (CAR X) ) T) (T (MEMBER A (CDR X)))) ))
))
```

The same program written in Common Lisp today would look like this:

```
(defun member (element list)
  (cond ((null list) ())
        ((eq element (first list)) t)
        (t (member element (rest list)))))
```

Notice the modern indentation is designed to clarify the relations between significant portions of the code. The old style appears to have no design.

This intermediate example is derived from a 1966 coding style:

```
DEFINE((
  (MEMBER (LAMBDA (A X) (COND
    ((NULL X) F)
    ((EQ A (CAR X) ) T)
    (T (MEMBER A (CDR X))) )))
))
```

The design of this style appears to take the name of the function, the arguments, and the very beginning of the `COND` as an idiom, and hence they are on the same line together. The branches of the `COND` clause line up, which shows the structure of the cases considered.

With MacLisp, the style of indentation was consciously evolved, because programmers themselves were responsible for doing the indentation, at least until text editors like Emacs [?] added semi-automatic indentation facilities. Program code was viewed by text editors and on printout, while in Interlisp program code was primarily, but not exclusively, viewed on the screen.

Interlisp provided a pretty-printer that would take the in-memory program and dump it to a file or to the terminal. The structure editor used the pretty-printer to show program code. Though the pretty-printer was crudely programmable (???) few people customized it. (Fonts???) Because MacLisp had a pretty-printer and programmers had adopted their own pretty-printing style, the pretty-printer had to be programmable, and it was. [White, 1969–1982; Moon, April 1974; Pitman, 1983; ?; Steele, 1990c]

Of course, Interlisp developers improved Interlisp, but the primary language improvements were spaghetti stacks (and hence a sort of dynamic closure), the record package, and field-based structure. In terms of performance, Interlisp developers added block compilation and using more than one PDP-10 address space.

However, the primary mechanisms for language extensions were through the environment. DWIM, the spelling corrector, and CLISP served to define a new language by changing the surface syntax and tolerating certain types of errors.

As a result, there were fewer language experiments and more environment experiments in Interlisp, while the opposite was true in MacLisp. These two dialects were the primary dialects throughout the 1970's and into the 1980's, yet when the big consolidation of the early 1980's took place, MacLisp and MacLisp derivatives provided by far the strongest influences. This is due to the stagnation of the core Lisp language within Interlisp. This stagnation was the price of environmental attention by the Interlisp developers.

Of course, there were other factors in Interlisp's demise, not the least of which was the belief by its champions that Interlisp was enjoying excellent health and new prospects up to the very end.

Interlisp also represented a consolidation after diversification and acceptance, and its later diversification activities centered around the environment and not the core language. Unlike MacLisp, the developers of Interlisp tried to spread Interlisp through porting it to other machines, while the MacLisp developers—each off at new institutions and trying out new language ideas—were diversifying by making new designs and new variants of Lisp.

2.4 The Early 1970's

Though MacLisp and Interlisp dominated the 1970's, there were several other major Lisp dialects in use during this period. Most were more similar to MacLisp than to Interlisp. The two most widely used dialects were Standard Lisp [?][Marti, 1979] and Portable Standard Lisp [Group, 1982]. Standard Lisp was defined in 1966 [Griss, 1982]) by Anthony Hearn and Martin Griss, along with their students and colleagues. The motivation was to define a subset of Lisp 1.5 and other Lisp dialects that could serve as a medium for porting Lisp programs, most particularly the symbolic algebra system REDUCE.

Standard Lisp was designed so that if an existing Lisp implementation (called the “target Lisp”) could implement the Standard Lisp constructs, then Standard Lisp and REDUCE could run on top of that existing dialect.

Later Hearn and his colleagues discovered that for good performance they needed more control over the environment and the compiler, and Portable Standard Lisp (PSL) was born. Standard Lisp attempted to piggyback on existing Lisps, while PSL was a complete, new Lisp implementation with a retargetable compiler [Griss, 1981], an important pioneering effort in the evolution of Lisp compilation technology. By the end of the 1970's, PSL ran on more than a dozen different types of computers.

PSL was implemented using two techniques. First, it used a system implementation language called SYSLISP, which was used to code operations on raw, untyped representations. Second, it used a parameterized set of assembly-level translation macros called c-macros. The Portable Lisp Compiler (PLC) compiled Lisp code into an abstract assembly language. This language was then converted to a machine-dependent LAP (Lisp Assembly Program) format by pattern-matching the c-macro descriptions with the abstract instructions in context. For example, different machine instructions might be selected depending on the sources of the operands and the destination of the result of the operation.

In the latter half of the 1970's and on into the mid-1980's, the PSL environment was improved by adapting editors and other tools. In particular, a good multiwindow Emacs-like editor called Emode was developed that allowed fairly intelligent editing and the passing of information back and forth between the Lisp and the editor. Later, a more extensive version of Emode called Nmode was developed by Martin Griss and his colleagues at Hewlett-Packard in Palo Alto, California. This version of PSL and Nmode was commercialized by HP in the mid-1980's.

At Stanford in the 1960's, an early version of MacLisp was adapted to the PDP-6; this Lisp was called Lisp 1.6 [Quam, 1972]. The early adaptation was rewritten by John Allen and Lynn Quam; later compiler improvements were made by Whit Diffie.

UCI Lisp [Bobrow, 1972] was an extended version of Lisp 1.6 in which an Interlisp style editor and other programming environment improvements were made. UCI Lisp was used by some folks at Stanford during the early to mid-1970's, as well as at other institutions.

In 1976 the MIT version of MacLisp was ported to the WAITS operating system by Richard Gabriel at the Stanford AI Laboratory (SAIL), which was directed at that time by John McCarthy. This dialect supplanted the Lisp 1.6 derivatives, including UCI Lisp. At the Heuristic Programming Project, under the direction of Edward Feigenbaum, Interlisp was the primary dialect. Cordell Green's automatic programming group (PSI) used Interlisp via remote login to a PDP-10 at Information Science Institute (ISI) in southern California.

Analysis

Standard Lisp and PSL represented a consolidation of ideas with the primary focus of delivering a particular program, Reduce, to a range of users who used among them a variety of Lisp dialects. The users of Reduce were on the rise and eager to use it. These two dialects were seen as necessary for the spread of the useful Reduce system. Standard Lisp was an attempt to piggyback on existing Lisps, while PSL represented an attempt to control performance a little better.

UCI Lisp was an adaptation of Interlisp to a simple Lisp 1.6 base, and so we can regard this as part of the consolidation of Interlisp.

Lisp 1.6 itself disappeared during the mod-1970's, it being one of the last remnants of the Lisp 1.5 era.

2.5 The Demise of the PDP-10

By the middle of the 1970's it became apparent that the 18-bit address space of the PDP-10 would not provide enough working space for AI programs. The PDP-10 line of computers (KL-10's and DEC-20's) was altered to permit an extended addressing scheme, in which multiple 18-bit address spaces could be addressed by indexing relative to 30-bit base registers.

However, this addition was not a smooth expansion to the architecture as far as the Lisp implementor was concerned; the change from two pointers per word to only one pointer per word required a complete redesign of nearly all internal data structures. Only two Lisps were implemented for extended addressing: ELISP by Charles Hedrick at Rutgers [?] and PSL.

One response to the address space problem was to construct special-purpose Lisp machines (see section 2.6). The other response was to use commercial computers with larger address spaces; the first of these was the VAX [DEC, 1981].

Vaxen presented both opportunities and problems for Lisp implementors. The VAX instruction set provided some good opportunities for implementing the low level Lisp primitives efficiently, though it required clever—perhaps too clever—design of the data structures. However, Lisp function calls could not be accurately modeled with the VAX function-call instructions. Moreover, the VAX, despite its theoretically large address space, was apparently designed for use by many small programs, not several large ones. Page tables occupied too large a fraction of memory, and paging overhead for large Lisp programs was a problem never fully solved on the VAX. Finally, there was the problem of prior investment; more than one major Lisp implementation at the time had a large assembly-language base that was difficult to port.

The final blow came when DEC abandoned the entire PDP-10 architecture to concentrate on the VAX. In retrospect this was undoubtedly a good strategic decision on DEC's part, but at the time it only gave those Lisp implementors who had loved the PDP-10 another reason to resent the VAX. A lucky few had the resources to design and build their own hardware; for the rest, the VAX was the only game in town. To the PDP-10 aficionados it seemed slow, expensive, and unlovable.

The primary Vax Lisp dialects developed in the late 1970's were VAX Interlisp, PSL (ported to the VAX), Franz Lisp, and NIL.

Franz Lisp [Foderaro, 1982] was written to enable research on symbolic algebra to continue at the University of California at Berkeley, under the supervision of Richard J. Fateman, who was one of the principal implementors of Macsyma at MIT. Fateman and his students started with a PDP-11 version of Lisp written at Harvard, and extended it into a MacLisp-like Lisp that eventually ran on virtually all Unix-based computers, thanks to the fact that Franz Lisp is written almost entirely in C. Later Franz, Inc., was formed to further Franz Lisp; development and sales of Franz Lisp continue to this day.

Because Franz was intended to be primarily a vehicle for research in symbolic algebra, it never became a solid Lisp in the same sense that MacLisp did. On the other hand, it was a widely available Lisp dialect on one of the most widely available computers in the AI community.

NIL [Burke, 1983], intended to be the successor to MacLisp, was designed by Jon L. White, Guy L. Steele Jr., and others at MIT, under the influence of Lisp-Machine Lisp, also developed at MIT. Its name was a too-cute acronym for “New Implementation of Lisp” and caused a certain amount of confusion. NIL was a large Lisp, and efficiency concerns were paramount in the minds of its MacLisp-oriented implementors; soon its implementation was centered around a large VAX assembly-language base.

In 1978, Gabriel and Guy Steele set out to implement NIL [Brooks, 1982a] on the S-1 Mark IIA, a supercomputer being designed and built by the Lawrence Livermore National Laboratory [Correll, 1979; Hailpern, 1979]. This Lisp was never completely functional, but served as a testbed for adapting advanced compiler techniques to Lisp implementation. In particular, the work generalized the numerical computation techniques of the MacLisp compiler and unified them with mainstream register allocation strategies [Brooks, 1982b]. With the development of the S-1 Lisp compiler, it once again became feasible to implement Lisp in Lisp and to expect similar performance to the best hand-tuned, assembly-language-based Lisp systems.

In 1975 Gerald J. Sussman and Guy L. Steele Jr. began experimenting with some interpreters in an attempt to understand the consequences of the Actor model of computation. (This experimentation is explained in section 2.8.) This led to the publication during 1976–1978 of a series of papers describing a new dialect of Lisp called Scheme.

Scheme was one of the first languages to have taken seriously the implications of lexical scoping (as opposed to local scoping, which had been in use in Lisp compilers for almost a decade) and first-class functions. Namely, Scheme correctly treated closures—a closure is a function along with the local environment within which it was defined. Furthermore, it embraced the Actor model of computation in the sense that function calling could be treated as a jump to a procedure rather than as a call to a procedure that would return.

The theory stated that the remainder of the computation was encapsulated in another function (or procedure as Scheme terminology has it) called a continuation. A continuation takes one argument, which is the value to be passed on (returned). This leads to a style of programming called *continuation-passing*. For example, adding 7 to the result of multiplying *x* and *y*, giving the result to the continuation *cont*, would be written like this:

```
(* x y (lambda (temp) (+ temp 7 cont)))
```

Thus every primitive function takes an additional argument, which is its continuation, and so, by convention, do user-defined functions.

In an interpreter setting, this argument can be implicit. When explicit, the definition of *** would look like this:

```
(define * (lambda (m1 m2 cont)
             (cont (*$ m1 m2))))
```

where **\$* is the low-level multiplication primitive.

Scheme slowly gathered a following of theoretically and elegance and simplicity minded individuals who wrote interpreters, compilers, and systems based on the three seminal Scheme papers. Other Lisp developers, taken with the clarity of the language and its presentation, would for years proclaim their preference for Scheme over the less clean language their acceptance group required.

At the end of the decade, Scheme had had at least one good compiler written for it [Steele, 1978a].

In France in the mid-1970's, Greussay [Greussay, 1977] developed an interpreter-based Lisp called Vlip. At the level of the base dialect of Interlisp, it introduced a couple of interesting concepts, such as the *chronology*, which is a sort of dynamic environment for implementing interrupts and environmental functions like **trace** and **step**, by creating different incarnations of the evaluator. Vlip's emphasis was on having a fast interpreter. The concept was to provide a virtual machine that was used to transport the evaluator. This virtual machine was at the level of assembly language and was designed for easy porting and efficient execution. The interpreter got a significant part of its speed from two things: a fast function dispatch using a function type space that distinguished a number of functions of different arity, and tail recursion removal. (Vlip was probably the first production quality Lisp to support general tail recursion removal. Other dialects of the time, including MacLisp, did tail recursion removal in certain situations only, in a manner not guaranteed predictable.) Vlip was the precursor to Le.Lisp, one of the important Lisp dialects in France and Europe during the 1980's; though the dialects were different, they shared some implementation techniques.

Analysis

The PDP-10 was designed in such a way that the early dialects of Lisp could be implemented quite well on them: CONS cells fit into one word because memory addresses were 18 bits and words were 36 bits, there were convenient halfword manipulation instructions, and the function calling mechanism was amenable to Lisp.

With the advent of the Vax this changed. There had no easy matches between Lisp data structures and Vax architectural facilities. Furthermore, the preferred Vax procedure call instructions did not fit well with the new style of MacLisp-like Lisp dialects: Each of these dialects supported a variable number of arguments, and the Vax instruction set supported well only function calls with a fixed number of arguments, though there were several ways to cobble together an effective function call implementation for Lisp. At a time when other languages such as C and FORTRAN were learning to talk to each other, however, this provided yet one more impediment to Lisp's joining the crowd.

At the end of the 1970's, no new commercial machines suitable for Lisp were on the horizon; it appeared that the Vax was all there was. Despite years of valiant support by Glenn Burke, Vax NIL never achieved widespread acceptance. Interlisp/VAX was a performance disaster. "General-purpose" workstations (i.e., those intended or designed to run languages other than Lisp) and personal computers hadn't quite appeared yet. To most Lisp implementors and users, the commercial hardware situation looked quite bleak.

But from 1974 onward there had been research and prototyping projects for Lisp machines, and at the end of the decade it appeared that Lisp machines were the wave of the future.

At this point we have the situation where the natural acceptance groups for MacLisp are on the decline, but the acceptance groups of Interlisp are on the ascendency; these groups are the commercially minded AI groups that are about to be incorporated. Interlisp runs on a set of machines that are accepted by the commercial acceptance groups, and it fits their style of work. The new Lisp machines (discussed in another section) for Interlisp have a decided cachet. Things are ripe for Interlisp to claim the right of consolidation, perhaps even winning over MacLisp users.

MacLisp is entering a phase of diversification with the various Lisp-Machine dialects. It appears that the Lisp machines are too new to be readily acceptable; and their new development environments are maybe too much of a change from the old-style MacLisp development to move into an acceptance phase. But they do have cachet.

The race will be to see who can choose the machine or machines with the right characteristics to help the Lisp succeed. The immediate question is whether the Vax will be the right machine, the Lisp machines (and which one), or a machine then unknown. It will turn out to be a then-unknown machine.

Notice that Franz Lisp and PSL have nothing to offer the new acceptance groups except for portability, and each as a Lisp is a throwback to earlier MacLisp times. Though portability appears to be an important survival characteristic, without some cachet there is little hope for a dialect.

Scheme represents an interesting variation on our cycle theme. Gerry Sussman has been involved with a variety of language groups: the MDL (or Muddle) group, the Micro-Planner group, and the Conniver group. After Scheme he would work on constraint languages. His target audience is himself along with those researchers who are involved in work related to Sussman's work or researchers who wish to use one of Sussman's languages.

Sussman's work always has cachet—from Micro-Planner to Conniver to Scheme—and so his work almost always gains acceptance, but the acceptance groups do not have strong presence, so these ideas do not always catch on.

Surprising is the exception of MDL: Sussman belonged to this group but MDL itself did not catch on, and at MIT it was not highly regarded. Nevertheless, a great many ideas from MDL caught on in Conniver and then Lisp-Machine Lisp. We still see these ideas hard at work in Common Lisp and to some extent in EuLisp. Yet how many people who use these languages know of MDL, or its prime designer Chris Reeves?

Thus, Sussman was during the 1970's a diversification generator whose acceptance/consolidation phases were quite short, given the small acceptance audience.

In the global context, Scheme would provide concepts that would be used during the consolidation phases of other dialects. When we look at Scheme in the 1980's we will see that there is one primary acceptance group for Scheme, along with the potential of some others. The primary acceptance group for Scheme is the set of authors of the various revised reports on Scheme.

2.6 Lisp Machines

The discussion of Lisp can be broken down into two categories according to the type of hardware on which the Lisp can run. The two important categories are *stock hardware* and *Lisp machines*. A stock hardware machine is one that is commercially designed for general purpose computing. For the purposes of our discussion a stock hardware computer can be taken to be a computer that is not designed specifically to support Lisp. For example, a Digital Equipment Corporation Vax 11/780 is stock hardware, and a Symbolics 3600 is an example of a computer designed specifically to support Lisp. Some Lisp machine designers refer to stock hardware computers as FORTRAN machines.

We use the term “stock hardware” where other people today might use the term “general-purpose computer.” The latter term might seem a little more appropriate because it is in contrast to “special-purpose hardware.” However, the proper contrast is between general-purpose hardware with stock capabilities and general-purpose hardware with additional capabilities. Several of the operations frequently performed while executing Lisp programs can be better performed or significantly sped up by special hardware. (The most frequently hardware-assisted operations performed by special hardware on Lisp machines are tagging, function calling, and garbage collection.) But a Lisp machine typically is a general-purpose computer that can also be used to run Fortran or C code, at least in principle.

The words of storage in a Lisp machine can be designed to have enough bits to support pointers directly—the address and the tag bits can both fit in a word of storage without losing any addressing bits. Instructions to check the tag of a pointer can be made fast with special data paths in the

computer. Operations, such as addition, can check the types of their operands in parallel with the operation itself, and if the types are not suitable, an exceptional condition can be flagged, and a more general course of action can then be taken.

Function calling is one of the most frequent operations in Lisp programs. The construction of stack frames can be performed by hardware. Caching parts of the stack can make a major performance difference to a Lisp implementation.

Garbage collection is often performed by need—when the dynamic heap runs out of free space a garbage collection can be initiated to gain some free space. Garbage collection can take a relatively long time, and performing it concurrently with other Lisp operations can eliminate the long pause associated with garbage collection. Most of the known techniques for *incremental garbage collection* are best implemented with special hardware.

Many Lisp machines are essentially general purpose computers which have been specially microcoded for Lisp.

Though ideas for a Lisp machine had been informally discussed before, Peter Deutsch seems to have published the first concrete proposal [Deutsch, 1973]. Deutsch outlined the basic vision of a single-user minicomputer-class machine that would be specially microcoded to run Lisp and support a Lisp development environment. The two key ideas from Deutsch’s paper that have had a lasting impact on Lisp are (1) the duality of load and store access based on functions and (2) the compact representation of linear lists through CDR-coding.

All Lisp dialects up to that time had one function `CAR` to read the first component of a dotted pair and a nominally unrelated function `RPLACA` to write that same component. Deutsch proposed that functions like `CAR` should have both a “load” mode and a “store” mode. If $(f\ a_1\ \dots\ a_n)$ is called in load mode, it should return a value; if called in store mode, as in $(f\ a_1\ \dots\ a_n\ v)$, the new value v should be stored in the location that would be accessed by the load version. Deutsch indicated that there should be two internal functions associated with every accessor function, one for loading and one for storing, and that the store function should be called when the function is mentioned in a particular set of special forms. However, his syntax is suggestive; here is the proposed definition of `RPLACA`:

```
(lambda (x y) (setfq (car x) y))
```

Deutsch commented that the special form used here is called `SETFQ` because “it quotes the function and evaluates everything else.” This name was abbreviated to `SETF` in Lisp-Machine Lisp. Deutsch attributed the idea of dual functions to Alan Kay.

2.6.1 MIT Lisp Machines: 1974–1978

Richard Greenblatt started the MIT Lisp Machine project in 1974; his proposal [Greenblatt, 1974] cites the Deutsch paper. The project also included Thomas Knight, Jack Holloway, and Pitts Jarvis. The machine they designed was called `CONS`, and its design was based on ideas from the Xerox PARC ALTO microprocessor, the DEC PDP-11/40, the PDP-11/40 extensions done by CMU, and some ideas on instruction modification suggested by Sam Fuller at DEC.

This machine was designed to have good performance while supporting a version of Lisp upwards-compatible with MacLisp but augmented with “Muddle-Conniver” argument declaration syntax. Its other goals included non-prohibitive cost (less than \$70,000 per machine), single user operation, common target language along with standardization of procedure calls, a factor of three better storage efficiency than the PDP-10 for compiled programs, hardware support for type checking and garbage collection, a largely Lisp-coded implementation (less reliance on assembly language or other low-level implementation language), and display-oriented interaction. , lengthy uptimes

(infrequent need to reboot the machine with to start Lisp), frame-oriented storage using a variety of techniques described below.

The CONS machine was to support invisible pointers, which are a generalization of the indirect pointer in the PDP-10. An indirect pointer was a memory storage format that had the following effect: If M is a memory location containing an indirect pointer to location T , then a special instruction that fetches from M will invisibly fetch from T , and a special instruction that stores into M will store into T . Normal memory instructions would store into and fetch from M .

Invisible pointers differ from this mechanism by switching the specification of such indirection from the instruction to the data, so that a special tag on the location M , for example, would signal that indirection to T would be used, not a special instruction.

Invisible pointers were used for the following things:

- Incremental garbage collection in which the operation of the garbage collector is interspersed with the operation of the Lisp system. In a copying collector, when an object is copied, a forwarding pointer that is an invisible pointer is left behind so that references to the old location would forward to the new one.
- CDR-coding, in which a linear, vector-like representation is used for lists rather than the pure CONS cell representations. In this representation each element of the vector contains the object logically stored in the CAR, and the element is tagged indicating that the next sequential cell is the CDR. When RPLACD is used on such a list, the old vector element is replaced with an invisible pointer that points to a real CONS cell instead of the more optimal representation. Thus the storage savings of CDR-coding can be maximally retained.
- Uniform treatment of special and local variables. The same addressing mode can be used to refer to special and local variables because the special variable can be represented as a location with an invisible pointer to the value cell.
- Locatives similar to those defined in MDL can be implemented providing an object to use for reference, where that object is an invisible pointer to the actual component.

The CONS machine was built; then a subsequent improved version named the CADR was designed and some dozens of them were built. The CADR was a specially microcoded, 32-bit processor, with some data paths of particular usefulness to Lisp. These became the computational mainstay within the MIT AI Lab, and it seemed sensible to spin off a company to commercialize this machine. Because of disagreements among the principals, however, *two* companies were formed: Lisp Machines Incorporated (LMI) and Symbolics. Initially each manufactured CADR clones. Soon thereafter Symbolics introduced its 3600 line, which became the industry leader in Lisp machine performance for the next 5 years.

While Greenblatt had paid particular care to providing hardware mechanisms to support fast garbage collection, the early MIT Lisp Machines in fact did not implement a garbage collector for quite some years; or rather, even when the garbage collector appeared, users preferred to disable it. Most of the programming tools (notably the compiler and program text editor) were designed to avoid consing and to explicitly reclaim temporary data structures whenever possible; given this, the Lisp Machine address spaces were large enough, and the virtual memory system good enough, that a user could run for several days or even a few weeks before having to save out the running “world” to disk and restart it. Such copying back and forth to disk was equivalent to a slow, manually triggered copying garbage collector. (While there was a great deal of theoretical work on interleaved and concurrent garbage collection during the 1970’s [?; ?; Baker, 1978; ?], continuous garbage collection was not universally accepted until David Moon’s invention of ephemeral garbage collection and its implementation on Lisp Machines [Moon, 1984]. Ephemeral garbage collection was subsequently adapted for use on stock hardware.)

The early MIT Lisp-Machine Lisp dialect [Weinreb, November 1978] was very similar to MacLisp. It lived up to its stated goal of supporting MacLisp programs with only minimal porting effort. The most important extensions beyond MacLisp included:

- An improved programming environment, consisting primarily of a resident compiler, debugging facilities, and a text editor. While this brought Lisp-Machine Lisp closer to the Interlisp ideal of a completely Lisp-based programming environment, it was still firmly file-oriented. The text editor was an EMACS clone, first called EINE (EINE Is Not EMACS) and then ZWEI (ZWEI Was EINE Initially), the recursive acronyms of course being doubly delicious as version numbers in German.
- Complex lambda lists, including `&optional`, `&key`, `&rest`, and `&aux`
- Locatives, which provided a C-like ability to point into the middle of a structure
- `DEFMACRO`, a much more convenient macro definition facility This is a macro definition facility that provides a mechanism for referring to parts of the form which will be passed to the macro function. See section 3.3.
- Backquote, a syntax for constructing data structures by filling in a template . This is a feature for constructing lists, and it derives from the MDL evaluation model and the Conniver list building facility.
- Stack groups, which provided a coroutine facility
- Multiple values, the ability to pass more than one value back from a function invocation without having to construct a list. Prior to this various ad hoc techniques had been used; Lisp-Machine Lisp was the first dialect of Lisp to provide primitives for it. (Other languages such as POP-2 have also provided for multiple values.)
- `DEFSTRUCT`, a record structure definition facility (compare the Interlisp record package)
- Closures over special variables. These closures were not like the ones in Scheme; the variables captured by the environment must be explicitly listed by the programmer and invocation of the closure required the binding of `SPECIAL` variables to the saved values.
- Flavors, an object-oriented, non-hierarchical programming system with multiple inheritance, was designed by Howard Cannon and David A. Moon and integrated into parts of the Lisp Machine programming environment (the window system, in particular, was written using Flavors [Weinreb, March 1981]).
- `SETF`, a facility for generalized variables

Consider the following sequence of forms passed to a Lisp machine Lisp interpreter. The last form returns 3 instead of 7, because the special variable `y` is reset to its old value by invoking the closure in `fun`, written in Lisp machine Lisp:

```
(defun foo (x)
  (+ (bar) x))

(defun bar () y)

(setq fun ((lambda (y)
             (closure '(x) (function foo)))
           3))
```

```
(setq y 7)
```

```
(funcall fun 0) => 3 ;not 7
```

The use of **SETF** throughout Common Lisp—a later and the most popular dialect of Lisp—can be traced through Symbolics Zetalisp and MacLisp to the influence of MIT Lisp-Machine Lisp and then back through Greenblatt’s proposal to Peter Deutsch and thence to Alan Kay.

The uniform treatment of access—reading and writing of state—has made Common Lisp more uniform than it might otherwise be. It is no longer necessary to remember both a reader function (such as **CAR**) and also a separate writer or update function (such as **RPLACA**), nor to remember the order of arguments (for **RPLACA**, which comes first, the dotted pair or the new value for its *car*?). If the general form of a read operation is `(f ...)`, then the form of the write is `(setf (f ...) newvalue)`, and that is all the programmer needs to know about reading and writing data.

In CLOS, this idea was extended to methods. If there is a method *M* that is specified to act as a reader and is invoked as `(M object)`, then it is possible to define a writer method that is invoked as `(setf (M object) newvalue)`.

That CLOS fits this idiom so well is no surprise. If Alan Kay was the inspiration for the idea around 1973, he was in the midst of his early Smalltalk involvement. Reading and writing are both methods that an object can support, and the CLOS adaptation of the Lisp version of Kay’s vision was a simple reinvention of the object-oriented genesis of the idea.

2.6.2 Xerox Lisp Machines: 1973–1980

The Alto was a microcodable machine developed in 1973 [Thacker, 1982] and used for personal computing experimentation at Xerox, using Interlisp and other languages such as Mesa [Geschke, 1977]. The Alto version of the Interlisp environment first went into use at PARC and at Stanford University around 1975.

The Alto was standardly equipped with 64K 16-bit words of memory, expandable up to 256K words, which was quite large for a single-user computer but still only half the memory of a PDP-10. The machine proved to be underpowered for the large Interlisp environment, even with all the code density tricks discussed by Deutsch in [Deutsch, 1973], so it was not widely accepted by users.

The Alto was also used to build the first Smalltalk environment—the “interim Dynabook”—and here it was relatively successful.

In 1976, Xerox Parc began the design of a machine called the Dorado (or Xerox 1132), which was an ECL (Emitter Coupled Logic, an at-the-time fast digital logic implementation technology) machine designed to replace the Alto. A prototype available in 1978 ran all Alto software. A redesign was completed in 1979 and a number of them were built for use within Xerox and at certain experimental sites such as Stanford University. The Dorado was specifically designed to interpret byte codes produced by compilers, and this is how the Dorado ran Alto software. The Dorado was basically an emulation machine.

Interlisp was ported to this machine using the Interlisp virtual machine model [Moore, 1976]. The Dorado running Interlisp was faster than a KL-10 running single-user Interlisp, and it would have proved a very nice Lisp machine if it had been made widely available commercially. the Dorado was designed as a prototype and was never able to pass the FCC RF emission requirements (??? check this is true), and its use was limited to Xerox sites and some special experimental settings, such as at Stanford University.

Interlisp was similarly ported to a smaller, cheaper machine called the Dolphin (1100), which was made commercially available as a Lisp machine in the late 1970’s. The performance of the Dolphin

was better than that of the Alto, but bad enough that the machine was never truly successful as a Lisp engine.

In the early 1980's, Xerox built another machine called the Dandelion (1108), which was considerably faster than the Dolphin but still not as fast as the Dorado. Because the names of these three machines all began with the letter "D", they became collectively known as the "D-machines."

All the Xerox Lisp machines used a reference-count garbage collector [Deutsch, 1976] that was incremental: a few steps of the garbage collection process would execute each time storage was allocated. Therefore there was a short, bounded amount of work done for garbage collection per unit time.

In the late 1970's BBN also built a machine, the Jericho, that was used as an Interlisp engine. It remained internal to BBN. [?]

Analysis of Early Lisp Machine History

The MIT Lisp machine project represented a consolidation of the diversification demonstrated by MacLisp, MDL, and Conniver. Some new extensions and some adaptations from other sources were made.

The acceptance group was still the local researchers, but during the 1980's that would change: the new acceptance group would become the commercial developer and corporate research lab. The acceptance battle would center around the factors of acceptable computer and acceptable work process. Included in the work process is the nature, cost, and availability of Lisp programmers.

The Xerox Lisp machines represented porting, which is what happens when the language is essentially stagnant. Acceptance of Interlisp would now hinge on its choice of machine. There would be two real contenders for the acceptable machine—the Vax and the D-machines. No Lisp whose success depended on the Vax would survive the 1980's. It would be a race between the D-machines, and the MIT-derived Lisp machines in the early 1980's, and the great consolidation started in the early 1980's would completely change the nature of the game. MacLisp and Interlisp would both be soon dead, and Lisp machine Lisp in its pure form along with them.

It is important to realize that the cachet of Lisp through the existence of Lisp machines was on the rise. The acceptance group would soon also be on the rise, and the feeling was that technology would triumph under the careful guidance of the Lisp machine designers and developers. Companies would form to ride the wave of AI and Lisp technology, consuming immense quantities of venture capital. This was the first—and until today the only—plausible attack mounted on fortress FORTRAN-machine. As we know that attack would fail, and discovering the reasons may teach us a lot about ourselves as computing professionals, and as people.

Freed from the address-space constraints of previous architectures, all the Lisp machine companies produced greatly expanded Lisp implementations, adding graphics, windowing capabilities, and mouse interaction capabilities to their programming environments. The Lisp language itself, particularly on the MIT Lisp Machines, also grew in the number and complexity of features. Though some of these ideas originated elsewhere, their adoption throughout the Lisp community was driven as much by the success and cachet of the Lisp machines as by the cachet of the ideas themselves.

Nevertheless, for most users the value lay ultimately in the software and not in its enabling hardware technology. The Lisp machine companies ran into difficulty in the late 1980's, perhaps because they didn't fully understand the consequences of this fact. General-purpose hardware eventually became good enough to support Lisp once again, and Lisp implementations on such machines began to compete effectively.

2.7 IBM Lisps: Lisp360 and Lisp370

Although the first Lisps were implemented on IBM computers, IBM faded from the Lisp scene during the late 1960's, for two reasons: better cooperation between MIT and DEC and a patent dispute between MIT and IBM.

In the early 1960's, Digital Equipment Corporation discussed with MIT the needs MIT had for computers, and features were added to help Lisp implementations. As on the 7094, each 36-bit word could hold two addresses to form a dotted pair, but on the PDP-10 each address was 18 bits instead of 15. The PDP-10 halfword instructions made `CAR`, `CDR`, `RPLACA`, and `RPLACD` particularly fast and easy to implement. The stack instructions and the stack-based function calling instructions improved the speed of of Lisp function calls. (The MIT AI Laboratory received the first—or second—PDP-6, and it was the lab's mainstay computing engine until it was replaced by its successor, the PDP-10.)

Moreover, in the early 1960's, IBM and MIT disputed who had invented core memory, and IBM insisted on enforcing its patents against MIT. MIT responded by declining to use IBM equipment as extensively as it had in the past. This provided further impetus to use DEC equipment instead, particularly for Lisp and AI.

Nevertheless, Lisp was implemented at IBM for the IBM 360 and called Lisp360. When the IBM 370 came out, Lisp370 implementation began. Lisp370 was later called Lisp/VM.

Lisp360 was basically a batch Lisp, and it was used fairly extensively for teaching in universities.

Lisp370 began with the definition of a core Lisp based on a formal semantics expressed in the SECD model [Landin, 1964]. This definition fit on one or two pages. The Lisp370 project was under the direction of Fred Blair (who developed the SECD definition) at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. Other members of the group included Richard W. Ryniker II, Cyril Alberga, Mark Wegman, and Martin Mikelsons; they served primarily themselves and a few research groups, such as the SCRATCHPAD group and some AI groups at Yorktown.

Lisp370 supported both special binding and lexical binding, as well as closures over both lexical and special variables, using a technique similar to spaghetti stacks. The implementation of special value cells took the form of a cache for the actual values. The value cell was paired with a validation cell that specified a branch of the control tree for which the binding was valid. When invalid, a search from the current stack frame was made to set the correct value and the validation. The algorithm for this was developed by Allen Brown (who had previously been a student of Sussman's at MIT) with some assistance from Richard Gabriel and William Scherlis in the summer of 1976. Jon L White spent calendar year 1977 at Yorktown Heights working on Lisp370 and then returned to MIT; his experience at IBM had some influence on subsequent MacLisp development and on the NIL dialect.

Lisp370 had an Interlisp-like programming environment written to operate on both hardcopy terminals and the ubiquitous 3270 character-display terminal. The Interlisp-10 model was ideal because it was developed with slow terminals in mind, and the half-duplex nature of most IBM mainframe interactions had a similar feel. During the summer of 1976, Gabriel wrote the first version of this environment as a duplicate of the environment he had written for MacLisp at the Stanford AI Lab. (this environment itself being a revision of one he had written for MacLisp at the Co-ordinated Science Lab at the University of Illinois in 1974). Later, Mark Wegman and his colleagues at Yorktown extended this environment to include screen-based—rather than line-based—interaction and editing. , though it still retained the s-expression based editor and some of the other essential characteristics of the more primitive system Gabriel put together.

Improvements were made to the underlying Lisp system, such as good performance and the

separation of the compilation and runtime environments (which was accomplished through the use of separate Lisp images).

Other Lisps later appeared on the 360/370 line of computers, including several Common Lisps. The first Common Lisp to appear on the IBM 370 was written by Intermetrics, featuring good compilation technology. However, the Lisp was not well-constructed and never made much of an impact in the Lisp world. Several years later, Lucid ported its Common Lisp to the 370 under contract to IBM. IBM withdrew its support of Lisp370 in the late 1980's in favor of Common Lisp.

Analysis

The Yorktown Heights Lisps were based on a diversification surrounding the local acceptance group at the T. J. Watson Research Laboratory. In particular, this group consisting of Fred Blair, Richard W. Ryniker II, Cyril Alberga, Mark Wegman, and Martin Mikelsons served primarily themselves and a few research groups, such as the Scratchpad group and some AI groups.

This group had visitors that influenced the direction of diversification. For example, Allen Brown who was at the MIT AI Lab worked at Yorktown from 1975 to 1978, Jon L White worked there during the 1977 calendar year. Richard Gabriel worked during the summer of 1976 on the Lisp370 programming environment (though he was physically at the IBM research center in San Jose, California).

Acceptance was local and noncontroversial, but consolidation along with diversification was determined by influence from employees who came from other environments. In addition to the ones listed, Mark Wegman was a more recent addition to the team than the others, and his arrival coincided with a broader vision of the programming environment than Gabriel introduced.

The 370 never had any cachet except as the natural target of commercialization. It would turn out that even with the immensely popular (for a time) Common Lisp, the 370 never figured as much of a player in the success of Lisp. No one from the natural acceptance group for a Lisp considered the 370 a proper machine with a usable environment, even though Lisp370 certainly provided a nice development environment.

This brings up an interesting point. The AI companies like Intellicorp, Inference, and Teknowledge all targeted the Fortune 500 companies, particularly those that used 370's as their workhorses. Each AI company predicted that its own fortunes would rise if they could only get their products on the 370. Yet, except for Aion, this never made much difference to the AI companies. Aion provided a more modest set of products, aimed slightly more directly at the needs of the customers.

Most of the other companies built large expert system shells, which could then be programmed to solve particular problems. These solutions usually required the aid of the AI companies to create.

One could argue that the Lisp-based AI solutions produced by the AI companies failed to have a major impact because the solutions were written in Lisp. But, had the solutions been adequate, the underlying language would have made no difference. Perhaps from the perspective of AI, AI itself did not have the cachet that the AI companies believed it should have, and so these sorts of solutions were not accepted.

Or, perhaps, the cachet was there, but is irrelevant. Cachet is leading edge, and some organizations are not interested in leading edge, but in the safe course. These companies buy IBM PC's, and IBM 370's as well. The cachet of the Macintosh does not affect them, and neither does the cachet of AI.

Remember, cachet is a relationship between a thing that has it, and a person who wants it.

2.8 Scheme: 1975–1980

The dialect of Lisp known as Scheme was originally an attempt by Gerald Jay Sussman and Guy Steele during Autumn 1975 to explicate for themselves some aspects of Carl Hewitt’s theory of *actors* as a model of computation. Hewitt’s model was object-oriented (and influenced by Smalltalk); every object was a computationally active entity capable of receiving and reacting to messages. The objects were called actors, and the messages themselves were also actors. An actor could have arbitrarily many *acquaintances*; that is, it could “know about” (in Hewitt’s language) other actors and send them messages or send acquaintances as (parts of) messages. Message-passing was the only means of interaction. Functional interactions were modeled with the use of continuations; one might send the actor named “factorial” the number 5 and another actor to which to send the eventually computed value (presumably 120).

Sussman and Steele had some trouble understanding some of the consequences of the model from Hewitt’s papers and language design, so they decided to construct a toy implementation of an actor language in order to experiment with it. Using MacLisp as a working environment, they decided to construct a tiny Lisp interpreter and then add the necessary mechanisms for creating actors and sending messages. The toy Lisp would provide the necessary primitives for implementing the internal behavior of primitive actors.

Because Sussman had just been studying Algol [Naur, 1963], he suggested starting with a lexically scoped dialect of Lisp. It appeared that such a mechanism would be needed anyway for keeping track of acquaintances for actors. This allowed actors and functions to be created by almost identical mechanisms. Evaluating a form beginning with the word `lambda` would capture the current variable-lookup environment and create a closure; evaluating a form with the word `alpha` would also capture the current environment but create an actor. Message passing could be expressed syntactically in the same way as function invocation. The difference between an actor and a function would be detected in the part of the interpreter traditionally known as `apply`. A function would return a value, but an actor would never return; instead, it would typically invoke a *continuation*, another actor that it knows about. Thus one might define the function

```
(define factorial
  (lambda (n)
    (if (= n 0) 1 (* n (factorial (- n 1))))))
```

or the equivalent actor

```
(define actorial
  (alpha (n c)
    (if (= n 0) (c 1) (actorial (- n 1) (alpha (f) (c (* f n)))))))
```

Note in this example that the values of `c` and `n`, passed in the message that invokes the outer `alpha` expression, become acquaintances of the continuation actor created by the inner `alpha` expression. This continuation must be passed explicitly because the recursive invocation of `actorial` is not expected to return a value.

Sussman and Steele were very pleased with this toy actor implementation and named it “Schemer” in the expectation that it might develop into another AI language in the tradition of Planner and Conniver. However, the ITS operating system had a 6-character limitation on file names and so the name was truncated to simply “Scheme” and that name stuck.

Then came a crucial discovery—one that, to us, illustrates the value of experimentation in language design. On inspecting the code for `apply`, once they got it working correctly, Sussman and Steele were astonished to discover that the codes in `apply` for function application and for

actor invocation were identical! Further inspection of other parts of the interpreter, such as the code for creating functions and actors, confirmed this insight: the fact that functions were intended to return values and actors were not made no difference anywhere in their implementation. The difference lay purely in the primitives used to code their bodies. If the underlying primitives return values, then the user can write functions that return values; if all primitives expect continuations, then the user can write actors. But the `lambda` and `alpha` mechanisms were themselves identical, and from this Sussman and Steele concluded that actors and closures were the same concept. Hewitt later agreed with this assessment, noting, however, that two types of primitive actor in his theory, namely cells (which have modifiable state) and synchronizers (which enforce exclusive access), cannot be expressed as closures in a lexically scoped pure Lisp without adding equivalent primitive extensions.

Sussman and Steele did not think any less of the actors model—or of Lisp—for having made this discovery; indeed, it seemed to them that Scheme still might well be the next AI language, capturing many of the ideas then floating around about data and control structure but in a much simpler framework. The initial report on Scheme [Sussman, 1975b] describes a very spare language, with a minimum of primitive constructs, one per concept. (Why take two when one will do?) There was a function constructor `lambda`, a fixpoint operator `labels`, a condition `if`, a side effect `aset`, a continuation accessor `catch`, function application, variable references, and not too much else. There was an assortment of primitive data structures such as symbols, lists, and numbers, but these and their associated operations were regarded as practical conveniences rather than theoretical requirements.

In 1976 Sussman and Steele wrote two more papers that explored programming language semantics using Scheme as a framework. *Lambda: The Ultimate Imperative* [Steele, 1976b] demonstrated how a wide variety of control structure ideas could be modeled in Scheme. Some of the models drew on earlier work by Peter Landin, John Reynolds, and others [Landin, 1965; Reynolds, 1972; Friedman, November]. The paper was partly tutorial in intent and partly a consolidated catalog of control structures. (This paper was also notable as the first in a long series of “Lambda: The Ultimate X” papers, a running gag that is as well-known in the Lisp community as the “X Considered Harmful” titles are in the broader programming-languages community.) *Lambda: The Ultimate Declarative* [Steele, 1976a] concentrated on the nature of `lambda` as a renaming construct; it also provided a more extensive comparison of Scheme and Hewitt’s PLASMA (see section 4), relating object-oriented programming generally and actors specifically to closures. This in turn suggested a set of techniques for constructing a practical compiler for Scheme, which this paper outlined in some detail. This paper was a thesis proposal; the resulting dissertation discussed a Scheme compiler called RABBIT [Steele, 1978a]. Mitchell Wand and Daniel Friedman were doing similar work at Indiana University at about this time [Wand, 1977], and they exchanged papers with Sussman and Steele during this period.

Steele also presented a paper for the ACM National Conference that discussed the implications of lambda calculus for tail-recursion and optimization of procedure calls [Steele, 1977d]. Its net effect on the implementations of languages outside the Lisp community appears in hindsight to have been completely negligible. However, one listener remarked to Steele how much he had enjoyed the choice of procedure names in the examples shown on the overhead transparencies. At that moment Steele was struck by the realization that not all the world uses `FOO` and `BAR` as standard nonce terminology.

Subsequently Steele and Sussman wrote a revised report on Scheme [Steele, 1978c]; the title of the report was intended as a tribute to Algol but in turn inspired another increasingly silly series of titles [Clinger, 1985a; Clinger, 1985b; Rees, 1986; ?]. Shortly thereafter they wrote an extended monograph, whose title was a play on *The Art of the Fugue*, illustrating numerous small Lisp

interpreters with variations. This was written mostly in Sussman's study at home on Saturdays. Steele would bicycle over in the morning, dodging the Boston traffic; then they would make a huge pot of tea and spend the afternoon alternately debating and then typing on a dial-up line to MIT. The monograph was never finished; only parts Zero, One, and Two were published [Steele, 1978b]. Part Zero introduced a tiny first-order dialect of Lisp modeled on recursion equations. Part One discussed procedures as data and explored lexical and dynamic binding. Part Two addressed the decomposition of state and the meaning of side effects. Part Three was to have covered order of evaluation (call-by-value versus call-by-name), and Part Four was intended to cover metalanguage, macro processors, and compilers. That these last two parts were never written is no great loss, since these topics were soon treated adequately by other researchers. While *The Art of the Interpreter* achieved some notoriety in the Scheme underground, it was rejected by an ACM journal.

A great deal in all these papers was not new; their main contribution was to bridge the gap between the models used by theoreticians (studying actors and the lambda calculus) and practitioners (Lisp implementors and users). Scheme made theoretical contributions in such areas as denotational semantics much more accessible to Lisp hackers; it also provided a usable operational platform for experimentation by theoreticians. There was no need for a centralized implementation group to support Scheme at a large number of sites or on a wide variety of machines. Like Standard Lisp but even smaller and simpler, Scheme could be put up on top of some other Lisp system in a very short time. Local implementations and dialects sprang up at many other sites (one good example is Scheme 311 at Indiana University [Fessenden, 1983; Clinger, 1984]); it was several years before anyone made a serious attempt to produce a portable stand-alone Scheme system. [?]

Extensive work on Scheme implementations was carried on at Yale and later at MIT by Jonathan Rees, Norman Adams, and others. This resulted in the dialect of Scheme known as T; this name was a good joke all around, since T was to Scheme approximately what the NIL dialect was to MacLisp. The goal was to be a simple dialect with an especially efficient implementation [Rees, 1982]:

T centers around a small core language, free of complicated features, thus easy to learn. ... we have refrained from supporting features that we didn't feel completely right about. T's omissions are important: we have avoided the complicated argument list syntax of Common Lisp, keyword options, and multiple functionality overloaded on single functions. It's far easier to generalize on something later than to implement something now that one might later regret. All features have been carefully considered for stylistic purity and generality.

The design of T therefore represented a conscious break not only from the Lisp tradition but from earlier versions of Scheme in places where Steele and Sussman had relied on tradition rather than cleaning things up. Names of built-in functions were regularized. T replaced the traditional -P suffix with universal use of the question mark; thus `numberp` became `number?` and `null` became `null?`. Similarly, every destructive operation had a name ending with an exclamation point; thus `nconc`, the MacLisp name for the destructive version of `append`, became `append!`. (Muddle [Galley, 1975] had introduced the use of question mark to indicate predicates and Sussman had used this convention over the years in some of his writing. Interlisp had had a more consistent system of labeling destructive operations than MacLisp, using the letter d as a prefix.)

T was initially targeted to the VAX (under both Unix and VMS) and to Apollo workstations. Most of the system was written in T and bootstrapped by the T compiler, TC. The evaluator and garbage collector, in particular, were written in T and not in machine language. The T project started with a version of the S-1 Lisp compiler [Brooks, 1982b] and made substantial improvements; in the course of their work they identified several bugs in the S-1 compiler and in the original report

on RABBIT [Steele, 1978a]. Like the S-1 Lisp compiler, it relied heavily on optimization strategies from the mainstream compiler literature, most notably the work by Wulf and others on the BLISS-11 compiler [Wulf, 1975].

A second generation of T compiler, called ORBIT [Kranz, 1986], integrated a host of mainstream and Lisp-specific optimization strategies, resulting in a truly production-quality Scheme environment. RABBIT was organized around a principle of translating Lisp code by performing a source-to-source conversion into “continuation-passing style” (CPS); ORBIT generalized and extended this strategy to handle assignments to variables. The register allocator used trace scheduling to optimize register usage across forks, joins, and procedure calls. ORBIT also supported calls to procedures written in languages other than Lisp. (This was contemporaneous with efforts at CMU and elsewhere to develop general “foreign function call” mechanisms for Common Lisp.)

2.9 Prelude to Common Lisp: 1980–1984

In the Spring of 1981, the situation was as follows. Two Lisp machine companies had sprung up from the MIT Lisp machine project: Lisp Machines Inc. (LMI) and Symbolics, Inc. The former was founded principally by Richard Greenblatt and the latter by a larger group including David A. Moon. Both initially productized the CADR, the second MIT Lisp machine, and each licensed the Lisp machine software from MIT under an arrangement that included passing back any improvements made by the companies. Symbolics soon embarked on designing and building a follow-on Lisp machine, which it would call the 3600. The language Lisp-Machine Lisp had evolved greatly since the first definition published in 1978, acquiring a variety of new features, most notably an object-oriented extension called Flavors.

The Xerox Lisp machines Dolphin and Dorado were running Interlisp and were in use in research Labs mostly located on the West Coast. BBN was constructing its Interlisp machine called the Jericho, and a port of Interlisp to the Vax was under way.

At MIT a project had started to define and implement a descendant of MacLisp called NIL on the Vax and S1 computers.

At CMU, Scott Fahlman and his colleagues and students were defining and implementing a MacLisp-like dialect of Lisp called Spice Lisp, which was to be implemented on the Spice machine—Scientific Personal Integrated Computing Environment.

2.10 Early Common Lisp

If there were no consolidation in the Lisp community at this point, Lisp might have died. ARPA was not interested in funding a variety of needlessly competing and gratuitously different Lisp projects. And there was no commercial arena—yet.

2.10.1 Out of the Chaos of MacLisp

In April 1981, ARPA called a “Lisp Community Meeting”, in which the implementation groups got together to discuss the future of Lisp. ARPA sponsored a lot of AI research, and their goal was to see what could be done to stem the tide of an increasingly diverse set of Lisp dialects in its research community.

The day before the ARPA meeting, part of the Interlisp community got together to discuss how to present a situation of a healthy Interlisp community on a variety of machines. The idea was to push the view of a standard language (Interlisp) and a standard environment existing on an ever-increasing number of different types of computers.

The day of the meeting, the Interlisp community successfully presented themselves as a coherent group with one goal and mission.

The MacLisp-descended groups came off in a way that can be best demonstrated with an anecdote. Each group stood up and presented where they were heading and why. Some questions arose about the ill-defined direction of the MacLisp community in contrast to the Interlisp community. Scott Fahlman said, “the MacLisp community is *not* in a state of chaos. It consists of four well-defined groups going in four well-defined directions.” There was a moment’s pause for the laughter to subside [Steele, 1982].

Gabriel attended the Interlisp pow-wow the day before the ARPA meeting, and he also witnessed the spectacle of the MacLisp community at the meeting. He didn’t believe that the differences between the MacLisp groups were insurmountable, so he began to try to sell the idea of some sort of cooperation among the groups.

First he approached Jon L White. The idea they discussed was two-fold: to try to unify the MacLisp community’s Lisp aspirations, and to do so for ARPA under contract. At this time Gabriel was in the S1 Lisp project, which was implementing NIL, and thus there was a natural affinity with White. They discussed the idea with Nils Nilsson who was head of the AI Lab at SRI, and they agreed to try to pull together more support for the idea.

Second, Gabriel and White approached Guy Steele, then at CMU and affiliated with the SPICE Lisp project. The three of them—Steele, White, and Gabriel—were all associated one way or another with the S-1 NIL project, and they were on their way to Lawrence Livermore National Laboratory the day after the ARPA meeting. They discussed the collaboration in Gabriel’s un-air-conditioned VW bug while on the hot, dusty, long road to LLNL, with the windows down. They agreed to approach Fahlman.

A few months later, Gabriel, Steele, White, Fahlman, William Scherlis (a colleague of Gabriel’s then at CMU), and Rod Brooks (part of the S1 Lisp project) met at CMU, and some of the technical details of the new Lisp were discussed. In particular, the new dialect was to have the following basic features:

- Lexical scoping, including full closures
- Multiple values, of a form like Lisp-Machine Lisp, but perhaps with some modifications for single-value-forcing situations
- A Lisp-2, in the sense of separate value and function cells [Gabriel, 1988] (see section ??)
- Defstruct
- SETF
- Fancy floating point numbers, including complex and rational numbers (this was the primary influence of S-1 Lisp)
- Complex lambda-list declarations, similar to Lisp-Machine Lisp
- No dynamic closures (called “flexures” in NIL)

After a day and a half of technical discussion, this group went off to the Oakland Original, a greasy submarine sandwich place not far from CMU. During and after lunch the topic of the name for the Lisp came up, and such obvious names as NIL and SPICE Lisp were proposed and rejected—as giving too much credit to one group and not enough to others—and such non-obvious names as Yu-Hsiang Lisp were also proposed and reluctantly rejected.

The name felt to be best was “Standard Lisp,” but another dialect was known by that name already. In searching for similar words, the name “Common Lisp” came up. Gabriel remarked that

this wasn't a good name because we were trying to define an Elitist Lisp, and "Common Lisp" sounded too much like "Common Man Lisp."

The naming discussion resumed at dinner at the Pleasure Bar, an Italian restaurant in another Pittsburgh district, but no luck was had by all.

Later in E-mail, Moon referred to "whatever we call this common Lisp," and this time, amongst great sadness and consternation that a better name could not be had, it was selected.

The next step was to contact more groups. The key was the Lisp machine companies, which would be approached last. In addition, Gabriel volunteered to visit the Franz Lisp group in Berkeley and the PSL group in Salt Lake City.

Gabriel's meeting with the PSL group was cordial and informative, and the group agreed to participate in the discussions and to possibly consider the result, but it was clear they felt that the Common Lisp effort would be difficult to pull off, and that PSL had a clear head start as the next Lisp, given that it ran on so many machines. Nevertheless Don Morrison of the PSL group would become an active participant in the process. Martin Griss separately commented (personal communication to Guy Steele, January 21, 1982, likely a form letter with personalized salutation, as it accompanied a draft of the PSL manual):

We are torn between keeping PSL to be just a simple extension of Standard LISP, and the other extreme of implementation of an entirely new, more modern LISP. Interaction with the developers of the new Common LISP dialect (from the LISP Machine, Spice LISP, NIL, and S-1 community), and possible participation with this effort may change to balance somewhat. For the moment, we have been adapting or developing modules for PSL primarily based on the LISP Machine and Common LISP manuals, Franz LISP, and UCI LISP.

The meeting with the Franz group was less cordial, and it was clear that there was no strong interest, though a member of the group joined the balloting process to be described.

In a fit of optimism that usually does not characterize him, Gabriel approached the Xerox Interlisp group, and received a warmer welcome than he did in Berkeley, but he left with only an indication that someone, probably Bill VanMelle, would attend the meetings, mostly as an observer.

For no good reason, the Lisp370 group was never invited to join the Common Lisp Group, as it became known. Because the purpose seemed so clearly domestic—the interchange of software within the ARPA community—no European group would be contacted at all, and this would be seen as an unforgivable insult in later years.

Gabriel spent some time talking to ARPA about the plans for Common Lisp, and enlisted their support. ARPA allowed some of the funds that it was spending on the Spice project to be spent on Common Lisp, and they allowed the ARPANet to be used for the discussions, even providing accounts and net access to people who would not otherwise have been able to participate.

The people and groups engaged in this grassroots effort were, by and large, on the ARPANET—because they were affiliated or associated with AI labs—so it was natural to decide to do most of the work over the network through electronic mail, which was automatically archived. In fact this was the first major language standardization effort carried out nearly entirely by E-mail.

During the summer of 1981, Brooks and Gabriel were working at CMU with Steele on S-1 Lisp. Because of this coincidence, many meetings and a lot of work on Common Lisp could take place.

The meeting with Symbolics and LMI took place at Symbolics in June 1981. Steele and Gabriel drove from Pittsburgh to Cambridge for the meeting. The meeting alternated between a deep technical discussion of what should be in the dialect and a political discussion about why the new dialect was a good thing. From the point of view of the Lisp machine companies, the action was with Lisp machines, and the interest in the same dialect running more places seemed academic. Of

course, there were business reasons for getting the same dialect running in many places, but people with business sense did not attend the meeting.

At the end, both Lisp machine companies decided to join the effort, and the Common Lisp Group was formed:

Alan Bawden	Richard P. Gabriel	William L. Scherlis
Rodney A. Brooks	Joseph Ginder	Richard M. Stallman
Richard L. Bryan	Richard Greenblatt	Barbara K. Steele
Glenn S. Burke	Martin L. Griss	Guy L. Steele Jr.
Howard I. Cannon	Charles L. Hedrick	William vanMelle
George J. Carrette	Earl A. Killian	Walter van Roggen
David Dill	John L. Kulp	Allan C. Weschler
Scott E. Fahlman	Larry M. Masinter	Daniel L. Weinreb
Richard J. Fateman	John McCarthy	Jon L White
Neal Feinberg	Don Morrison	Richard Zippel
John Foderaro	David A. Moon	Leonard Zubkoff

As a compromise, it was agreed that it was worth defining a family of languages in such a way that any program written in the language defined would run in any language in the family. Thus, a sort of subset was to be defined, though it wasn't clear anyone would implement the subset directly.

Some of the Lisp machine features that were dropped were Flavors, window systems, multiprocessing (including multitasking), graphics, and locatives.

During the summer, Steele worked on an initial Common Lisp manual based on the Spice Lisp manual. His initial work was assisted by Brooks, Scherlis, and Gabriel. Scherlis provided specific assistance with the type system, mostly in the form of informal advice to Steele. Gabriel and Steele regularly discussed issues because Gabriel was living at Steele's home.

The draft, called the Swiss Cheese Edition—because it was full of large holes—was partly a ballot in which various alternatives and yes-no questions were proposed. Through a process of E-mail-based discussion and voting, the first pass of voting took place. This was followed by a face-to-face meeting in November of 1981, where the final decisions on the questions posed were settled.

This led to another round of refinement with several other similar drafts/ballots.

Most of the work was carried out by E-mail discussion. These discussions were captured in file archives. One research at the Sloan School at MIT has used these archives to study computer-mail-based decision-making and interaction.

This style of working had some interesting results. People are more prone to be insulting and forthright in E-mail than in person. Many of the discussions carried on over the network were quite spirited, but they had the advantage of being written down automatically. There was no need to rely on dim memory or pale ink. All the mail was automatically stored in a centralized place, so there was no chance of losing or misplacing it.

The discussions were often in the form of proposals, discussions, and counterproposals. Code examples from existing software or proposed new syntax were often exchanged. And all was subject to quick review by people wishing to come up to speed or to go back to the record.

This style also had some drawbacks. Foremost was that it was not possible to observe the reactions of other people, to see whether some point angered them, which would mean the point was important to them. There was no way to see that an argument went too far or had little support. This meant that time was wasted and that carefully crafted written arguments were required to get anything done.

Once the process began, the approach to the problem changed from just a consolidation of existing dialects, which was the obvious direction to take, to trying to design the right thing. Some people took the view that this was a good time to rethink some issues and to abandon the goal of strict MacLisp compatibility, which was so important to the early Lisp-Machine Lisp designs. Some issues, like whether NIL is both a symbol and a CONS cell, were not rethought, though it was generally agreed that they should be.

One issue that came up early on is worth mentioning, because it is at the heart of one of the major attacks on Common Lisp, which was mounted during the ISO work on Lisp (see section 2.12). This is the issue of modularization, which had two aspects: (1) whether Common Lisp should be divided into a core language plus modules and (2) whether there should be a division into the so-called white, yellow, and red pages. These topics appear to have been blended in the discussion.

“White pages” refers to the manual proper, and anything that is in the white pages must be implemented somehow by a Lisp whose developers claim it is a Common Lisp. “Yellow pages” refers to implementation-independent packages that can be loaded in, for example, TRACE and scientific subroutine packages. The “red pages” were intended to describe implementation-dependent routines, such as device drivers.

Nevertheless, the first question is brought up by a direct reading of the issue: Division of Common Lisp into a core plus modules.

If this were taken to mean a proposal that would have partitioned the language into layers with a central layer and outer layers that depend on the inner ones, then Common Lisp could have been more easily subsetted, which would have led to obvious implementations on smaller machines. This would have satisfied the need to cheap, prolific implementations. This would also have made providing educational versions of the language more readily available. It also would have prevented the strong attack during the ISO meetings by Europe and by, to a lesser degree, Japan.

The response from influential members is revealing: “This seems weird. Motivate it. Maybe these modules are optional at the implementation’s choice?” “Keeping things modular is a good goal, but don’t expect to succeed completely.” “The division only makes a little sense.” [?; ?]

The group focussed too much on the funny white-yellow-red distinction and not on the core-language/extended-language distinction. Had this gone differently, so would have the future of Common Lisp.

Carefully deferred was the decision regarding whether () was a symbol. Even though this decision was left until nearly the end of the decision process—causing people to emotionally accept Common Lisp and attach part of their egos to it—when the discussion came up, it was divisive. Symbolics threatened to withdraw from the group unless their position was accepted, and so it was. The salient paragraph from their message is as follows:

We have had some internal discussions about the T and NIL issues. If we were designing a completely new language, we would certainly rethink these, as well as the many other warts (or beauty marks) in Lisp. (We might not necessarily change them, but we would certainly rethink them.) However, the advantages to be gained by changing T and NIL now are quite small compared to the costs of conversion. The only resolution to these issues that Symbolics can accept is to retain the status quo.

This shows that there are some issues, apparently trivial, that can have a profound influence on people.

Three more drafts were made—the Colander Edition (July 29, 1982), the Laser Edition (November 16, 1982), and the Mary Poppins Edition (???). The cute names are explained by their subtitles:

Colander: Even More Holes Than Before—But They’re Smaller!

Laser Edition: Supposed to be Completely Coherent

Mary Poppins Edition: Practically Perfect in Every Way

Virtually all decisions were completed by early 1983, but it was almost a year before the book *Common Lisp: The Language* would be available, even with a fast publishing job by Digital Press.

During the period of Common Lisp design, Guy Steele took a leave of absence from CMU and joined Tartan Laboratories.

The declared goals of the Common Lisp Group were as follows, paraphrased from CLt11 [Steele, 1984]:

- **Commonality:** Common Lisp originated in an attempt to focus the work of several implementation groups, each of which was constructing successor implementations of MacLisp for different computers. While the differences among the several implementations will continue to force some incompatibilities, Common Lisp should serve as a common dialect for these implementations.
- **Portability:** Common Lisp should exclude features that cannot be easily implemented on a broad class of computers. This should serve to exclude features requiring microcode or hardware on one hand as well as features generally required for stock hardware, for example declarations. (We use the term *stock hardware* to describe commercially available, general-purpose, “off-the-shelf” computers as opposed to computers specifically designed to support Lisp.)
- **Consistency:** The interpreter and compiler should exhibit the same semantics.
- **Expressiveness:** Common Lisp should cull the best experience from a variety of dialects, including not only MacLisp but Interlisp.
- **Compatibility:** Common Lisp should strive to be compatible with Zetalisp, MacLisp, and Interlisp, in that order.
- **Efficiency:** It should be possible to write an optimizing compiler for Common Lisp.
- **Power:** Common Lisp should be a good system-building language, suitable for writing Interlisp-like user-level packages, but it will not provide those packages.
- **Stability:** Common Lisp should evolve slowly and with deliberation.

Spice Lisp provided two significant contributions to the Common Lisp effort. One is the draft Common Lisp manual, which was derived from the Spice Lisp manual (and also borrowed heavily from MIT Lisp Machine Manuals) and the other is a reference implementation. For several years after the appearance of CLtL I, Spice Lisp served as the source, literally, for a number of Common Lisp implementations. Computer companies would take the free source code and port it to their computers, providing a base level Common Lisp.

With time companies would prefer to either write their own Common Lisps from scratch or contract with a commercial Common Lisp company to gain expertise in the former case and maintenance in the latter.

Later will see that the development of CLOS benefited from the existence of a reference implementation that was freely available.

2.10.2 Early Rumbblings

The Common Lisp definition process was not all rosy. Throughout there was a feeling among some that they were being railroaded or that things were not going well. The Interlisp group had input in the balloting process, but at one point they wrote:

The Interlisp community is in a bit of a quandary about what our contribution to this endeavor should be. It is clear that Common Lisp is not going to settle very many languages features in Interlisp's favor. What should we do?

[?; ?]

Part of the problem was the strength of the Lisp machine companies and the need for the Common Lisp Group to keep them within the fold, which bestowed on them a particularly strong brand of power. On this point, one of the people in the early Common Lisp Group put it:

Sorry, but the current version [draft] really gives a feeling of 'well, what's the largest subset of Lisp-Machine Lisp we can try to force down everyone's throat, and call a standard?'

[?; ?]

The Lisp machine folks had a flavor of argument that was hard to contend with, namely that they had experience with large software systems, and in that realm the particular solutions they had come up with were, according to them, the right thing. The net effect was that Common Lisp grew and grew.

One would think that the voices of the stock machine crowd, who had to write a compiler for Common Lisp, would have objected, but the two strongest voices—Steele and Gabriel—were feeling their oats over their ability to write a powerful compiler to foil the complexities of Common Lisp. One often heard them, and later Moon, remark that a “sufficiently smart compiler” could solve a particular problem. Pretty soon the core group was quoting this “SSC” argument regularly. (Later, in the mouths of the loyal opposition, it became a term of mild derision.)

The core group eventually became the “authors” of CLtL I: Guy Steele, Scott Fahlman, David Moon, Daniel Weinreb, and Richard Gabriel. This group shouldered the responsibility for producing the language specification document and conducting its review. The self-adopted name for the group was the “Quinquevirate” or, more informally, the “Gang of Five”.

2.10.3 The Critique of Common Lisp

At the 1984 Symposium on Lisp and Functional Programming, Rod Brooks and Gabriel broke rank and delivered the stunning opening paper, “A Critique of Common Lisp” [Brooks, 1984]. This was all the more stunning because Gabriel and Brooks were founders of a company whose business plan was to become the premier Common Lisp company. Fahlman, on hearing the speech delivered by Gabriel, called it traitorous.

This paper was only the first in a string of critiques of Common Lisp, and many of those critiques quoted this paper. The high points of the paper reveal a series of problems that proved to plague Common Lisp throughout the decade.

The authors believed that the following should have been goals of the design:

Intellectual conciseness: A language should be small enough for an average programmer to keep in his head.

Compiler Simplicity: It should be a modest job to write a compiler that produces efficient code for the language.

Runtime Efficiency: The object code should run fast.

Ease of Programming: It should be simple and pleasant to write efficient programs.

Brooks and Gabriel contrast these potential goals with what really happened during the Common Lisp design effort:

There was an underlying belief by the Common Lisp Group that everyone would have a big Lisp machine.

There was too much hope that microcoding could hide the cost of type dispatch.

The cost of invisible pointers, an apparently necessary ingredient for such things as displaced arrays, was ignored.

Too many language costs were dismissed with the “a sufficiently smart compiler” argument.

There was too much emphasis on existing implementations of inherently inefficient ideas—arrays and vectors are an example.

Too much effort was spent overextending transportation mechanisms, such as pathnames and potential numbers.

There was a religious belief that Lisp was the correct language for systems programming built into the design.

There was an implicit belief that transportation should be invisible, and that large complex pieces of code like text editors should move from Common Lisp to Common Lisp with little or no effort.

The spirit of compromise left too many crucial things undefined.

Common Lisp generic arithmetic makes it difficult for a stock hardware programmer to write efficient numeric code because of the number and nature of the required declarations.

There was an unnecessary explosion of arithmetic operations.

Function calling was too expensive.

The expected size and power of computers was exaggerated.

Some good ideas were taken too far, such as **FORMAT** and **SETF**.

Things that should have been libraries were included in the language, for example sequence functions.

This theme reappeared in the history of Common Lisp—the emergence of a number of “unCommon” Lisps, straining, perhaps, the tolerance of even the most twisted lovers of overused puns. Each unCommon Lisp proclaimed its better approaches to some of the shortcomings of Common Lisp.

Examples of Lisp dialects that officially or unofficially declare themselves “unCommon Lisps” are Lisp370, Scheme, EuLisp, and muLisp. This is clearly an attempt to distance themselves from the perceived shortcomings of Common Lisp, but, less clearly, their use of this term attests to the apparent and real strength of Common Lisp as the primary Lisp dialect. To define a Lisp as standing in contrast to another dialect is to admit the supremacy of that other dialect. (Imagine Ford advertising the Mustang as “the unCorvette”.)

More than any other single phenomenon, this behavior demonstrates one of the key ingredients of Lisp diversification: extreme—almost juvenile—rivalry between dialect groups.

We have already seen Lisp370 and Scheme. EuLisp is the European response to Common Lisp, developed as the lingua franca for Lisp in Europe. Its primary characteristics is that it is a Lisp-1 (no separate function and variable namespaces), has a CLOS-style (Common Lisp Object System) generic-function type object-oriented system integrated from the ground up, has a built-in module

system, and is defined in layers to promote the use of the Lisp on small, embedded hardware and educational machines. Otherwise, EuLisp is Common-Lisp like, with more commonality than disjointness. The definition of EuLisp took a long time; started in 1986, it wasn't until 1990 that the first implementation was available (interpreter-only) along with a nearly complete language specification. Nevertheless, the layered definition, the module system, and the object-orientedness from the start demonstrate that new lessons can be learned in the Lisp world.

This is a complex period. Clearly there is consolidation, and for the first time it is a conscious consolidation. The consolidation is driven by two acceptance groups: ARPA community (an altruistic effect) and the AI companies (a commercial effect). The altruistic group feels that community good is driving the effort, while it is really the buy-in of the Lisp machine companies that make it fly. At some level the CLG knows that the Lisp machine people have to be courted, so too many compromises with them are made.

Lisp machines provide the cachet that drives the effort. The AI groups and ARPA are on the ascendancy, so there is fuel for the effort.

Several new leaders emerge: Steele, a scholarly and sometimes naïve leader. Fahlman, a charismatic leader, especially of younger participants. Gabriel, the self-styled political mastermind who maneuvers ARPA and various groups into line. Moon, a seductively powerful thinker, quiet and often insulting, whose arguments are almost impossible to refute. Weinreb, the reflective listener who pays attention to every side of the argument. Old leaders like Jon L fade slightly into the background.

McCarthy shows little or no interest, but pays Gabriel all the while. This is typical of McCarthy, whose has little interest in details like this.

There is an emergence of new implementation leaders. Gabriel, Steele, Brooks, Fahlman. White disappears into Xerox and the Interlisp community.

Spice Lisp appeared as a reference implementation for Common Lisp, and several groups—TI, PERQ, Siemens, DEC—use it as the basis of their initial Common Lisps. Spice Lisp is freely available, and it sets the stage for free reference implementations for other dialects of Lisp. Furthermore, the appearance of a public domain reference implementation of the emerging Common Lisp aids its adoption.

Because of email, everyone who wants a voice in the design has one. Therefore, there is a group spirit that aids adoption of the ideas, because for the first time the common man is part of the design process, not the observer of an expert or group of experts. By merely asking for his or her name to be added to a mailing, anyone with ARPANET access can join the process. Though the Quinquvirate emerges as a expert group at the end (1983) to shepherd the final drafting of the definition.

Some groups don't see the power of Common Lisp coming. The Interlisp group is forging ahead, and rightly so because their acceptance group overlaps the Symbolics/LMI acceptance groups. The problem for InterLisp is that they have the wrong machines—the slow D-machines and the unpopular Vax. The Vax implementation of Interlisp never really got off the ground, partly for not being completed but mostly because performance of Interlisp is unacceptable. Of course, the problem lies more with the Vax than with Interlisp: the Vax was designed to run many small programs rather than one or two large ones, so the page size requires a two-level page table because the page table needs to be paged.

At the time the ARPA program director overseeing ARPA's Lisp concerns is Ron Ohlander, who is a powerful friend of Common Lisp. His friendliness is due in part to his perception that the fate of Lisp is central to the fate of AI, in part because the principals of AI tell him that he needs to pay attention to the underlying technology, and in part because the Common Lisp Group pays a lot of attention to him and his concerns. It probably doesn't hurt that it is on Ohlander's watch

that the contentious Lisp community finally gets its act together and agrees on a standard.

2.11 Other Stock Hardware Dialects: 1980–1984

The rest of the world did not stand still while Common Lisp was developed, though Common Lisp was the focus of a lot of attention. Portable Standard Lisp spread to the Vax, DEC-20, a variety of MC68000 machines, and the Cray 1. Its Emode environment (later Nmode) proved appealing to Hewlett-Packard, which “productized” it in the face of a growing Common Lisp presence.

Franz Lisp was ported to many systems, and it became the workhorse stock hardware Lisp for the years leading up to the general availability of Common Lisp in 1985–1986.

Interlisp gained acceptance from its pure acceptance group during the early 1980’s. It was running on PDP-10’s, Dolphins, Dorados, and Vaxes, and it seemed about to appear on a low-cost Lisp machine called the Dandelion, which was rumored to have Dorado-like performance for a Dolphin-like price.

In the market, the Dolphin was taking a beating in the performance sweepstakes, primarily because it was a slow machine that ran the Interlisp virtual machine. The efforts of Xerox were aimed at porting and performance, with little attention to improving the dialect or the environment, though work continued in this area. The main Interlisp developers were tuning.

Interlisp/Vax made an appearance, but has to be regarded as a failure with three contributing causes: (1) it provided compatibility with Interlisp-10, the branch of the Interlisp family doomed by the eventual demise of the PDP-10, rather than with Interlisp-D; (2) it provided only a stop-and-copy garbage collector, which has particularly bad performance on a Vax; and (3) as the rest of the Lisp world, including the Interlisp world, flocked to personal Lisp machines, the Vax was never taken seriously for Lisp purposes except by a small number of businesses.

In France, Jérôme Chailloux and his colleagues developed a new dialect of Lisp called *Le_Lisp*. The dialect was reminiscent of *MacLisp*, and it focused on portability and efficiency. It needed to be portable because the computer situation in Europe was not as clear as it was in the US for Lisp. In the US, Lisp machines were dominant for Lisp, but these machines were not as available and often were prohibitively expensive in Europe. Research labs in Europe were frequently given or acquired a range of peculiar machines (from the US perspective). Therefore, portability was a must.

The experience with *Vlisp* taught Chailloux that performance and portability can go together, and, extending some of the *Vlisp* techniques, his group was able to achieve their goals. By 1984 their dialect ran on about ten different machines and demonstrated performance very much better than *Franz Lisp*, the most comparable alternative. On Vax 780’s, *Le_Lisp* performed about as well as *Symbolics 3600*’s.

In addition, *Le_Lisp* provided a full-fledged programming environment called *Ceyx*. *Ceyx* had a full set of debugging aids, a full-screen, multi-window structure editor and pretty printer, and an object-oriented programming extension, also called *Ceyx*.

Of importance is that a subset of this Lisp was provided to French high schools for instruction by the French Education Ministry, providing a generation of *Le_Lisp* users.

Le_Lisp was quite portable, with some ports requiring only 1 week of effort by several people. The language was based on a virtual machine similar in philosophy to the one in which the *Vlisp* interpreter was written, and the compiler generated code for this machine, called *LLM3*. In addition to machine-independent optimizations, a peephole optimizer at load time was used.

Though *Le_Lisp* would not figure into Common Lisp history during this period, in the late 1980’s it would rise in prominence during the standardization wars.

The Scheme community grew from a few aficionados to a much larger group, characterized by an interest in the mathematical aspects of programming languages. Scheme's small size, roots in lambda calculus, and generally compact semantic underpinnings began to make it popular as a vehicle for research and teaching. In particular, strong groups of Scheme supporters developed at MIT, Rice, and Indiana University. In general these groups were started by MIT graduates who joined the faculty at these schools.

Though Scheme had continuations as a result of its roots in *Conniver* and *Actors*, the concept was generalized somewhat in the primitive `CALL-WITH-CURRENT-CONTINUATION`, which provided a means to capture the continuation at any point. With this primitive it was possible to explore such things as alternative models of multitasking, such as exemplified by the work at Indiana University on engines (??? refer to Haynes).

At MIT, under the guidance of Gerry Sussman and Hal Abelson, Scheme was adopted for teaching undergraduate computing. The book *Structure and Interpretation of Computer Programs* [?] became a classic and vaulted Scheme to notoriety in a larger community.

Several companies sprang up that made commercial implementations of Scheme: *Chez Scheme* by Cadence Research Systems was started by R. Kent Dybvig, *Semantic Microsystems* by Will Clinger, Anne Hartheimer, and John Ulrich, and Texas Instruments fielded a version of Scheme. *Chez Scheme* ran on various workstations, *Semantic Microsystems'* Scheme was called *MacScheme* and ran on Macintoshes. PC Scheme ran on IBM PC's and clones such as the one TI built and sold.

The original Revised Report on Scheme was taken as a model for future definitions of Scheme, and a self-selected group of so-called "authors" took on the role of evolving Scheme. The rule they adopted was that features could be added only by unanimous consent. After a fairly short period in which certain features like `CALL-WITH-CURRENT-CONTINUATION` were added, the rate of change of Scheme slowed down due to this rule. Only peer pressure in a highly intellectual group could convince any recalcitrant author to change his blackball. As a result there is a widely held belief that whenever a feature is added to Scheme, it is clearly the right thing. For example, only in late 1991 were macros added to the language in an appendix—as a partially standardized facility.

There would be a series of Revised Reports, called "The Revised, . . . , Revised Report on Scheme." In late 1991 "The Revised, Revised, Revised, Revised Report on Scheme" was written and approved; it is affectionately called "R⁴RS."

Many of those who would later become members of the Common Lisp Group would proclaim a deep-seated love of Scheme and a not-so-secret desire to see something like it become the next Lisp standard. However, the Scheme and Common Lisp communities would become sometimes bitter rivals in the latter part of the decade.

2.11.1 Zetalisp

At the time that Common Lisp began (1981) MIT Lisp-machine Lisp had evolved considerably. The Common Lisp Group (see below) needed to address Zetalisp's growing influence.

The name "Zetalisp" is chosen as (???) "the last word on Lisp."

Zetalisp was the name of the Symbolics version of Lisp-machine Lisp. Because the 3600—the Symbolics second-generation Lisp machine, which is described below—was programmed almost entirely in Lisp, Zetalisp came to require a significant set of capabilities not seen in any single Lisp to this point. Not only was Zetalisp used for ordinary programming, but the operating system, the editor, the compiler, the network server, the garbage collector, and the window system were all programmed in Zetalisp, and because the earlier Lisp-machine Lisp was not quite up to these tasks, Zetalisp was expanded to handle them.

The primary addition to Lisp-machine Lisp was Flavors, a so-called non-hierarchical object-oriented language. We will talk more about Flavors in the section (???), but for now suffice it to say that it is a multiple inheritance, message-passing system developed from some ideas of Howard Cannon. The development of the ideas into a coherent system was largely due to David A. Moon, though Cannon was a key individual.

The features of Flavors was driven by the needs of the Lisp machine window system, which, for a long time, was regarded as the only example of a system whose programming required multiple inheritance.

Other noteworthy additions were FORMAT and SETF, complex arrays, and complex lambda-lists for optional and keyword-named arguments. FORMAT is a mechanism for producing string output conveniently by, basically, taking a pre-determined string with placeholders and substituting computed values or strings for those placeholders—though it became much more complex than this because the placeholders included iteration primitives for producing lists of results, plurals, and other such exotica. SETF was discussed earlier.

One of the factors in the acceptance and importance of Zetalisp was the acceptance of the Lisp machines, which is discussed in the next section. Because Lisp machines—particularly Symbolics Lisp machines—were the most popular vehicles for real Lisp work in a commercial setting, there would grow to be an explicit belief, fostered by Symbolics itself, that Lisp-machine Lisp (Zetalisp) was the primary dialect for Lisp. Therefore, the Symbolics folks were taken very seriously as a strong political force and a required political ally for the success of a wider Lisp standard.

2.11.2 Early Lisp Machine Companies

There were five primary Lisp machine companies: Symbolics; Lisp Machine, Inc. (LMI); Three Rivers Computer, later renamed PERQ after its principal product; Xerox; and Texas Instruments (TI).

Of these, Symbolics, LMI, and TI all used basically the same software licensed from MIT as the basis of their offerings. The software included the Lisp implementation, the operating system, the editor, the window system, the network software, and all the utilities. There was an arrangement wherein the software would be cheaply (or freely) available as long as improvements are were passed back to MIT.

Therefore, the companies competed primarily on the basis of hardware performance but secondarily on the availability of advances in the common software base before that software passed back to the common source. Some of the companies also produced propriety extensions, such as C and FORTRAN implementations from Symbolics.

Xerox produced the D-machines, which ran Interlisp-D.

Three Rivers sold a machine, the PERQ, that ran either a Pascal-based operating system and language or Spice Lisp, later Common Lisp based on Spice Lisp.

Thus, all the Lisp-machine companies started out with existing software, and all the MacLisp-derived Lisp-machine companies licensed their software from a university.

Of these companies, Symbolics is most successful (as measured by number of installed machines at the end of the pre-Common-Lisp era), followed by Xerox and TI, though TI possibly can claim the most installed machines on the basis of one or two large company purchases.

Symbolics is most interesting of these companies because of the extreme influence of Symbolics on the direction of Common Lisp—however, we do not want to claim that the other companies did not have strong significance. The popularity of Zetalisp—or at least its apparent influence on the other people in the Common Lisp group—stemmed largely from the popularity of the 3600.

The 3600 was a second-generation Lisp machine, the first being a version of the CADR, called the LM-2. Both Symbolics and LMI started their businesses by producing essentially the CADR; the LMI version was called the LAMBDA. However, Symbolics's business plan was to produce a much faster Lisp machine and to enter the workstation sweepstakes.

Sometimes it is easy to forget that, in the early 1980's, workstations were an oddity, and the workstations were generally so computationally underpowered that many did not take them seriously. The PDP-10 still offered vastly better performance than the workstations, and it simply was not obvious that engineers would ever warm up to them or that they would form a large new market. Furthermore, it was not clear that a Unix-based/C-based workstation was necessarily the winner either, since it was thought that applications would drive the market more than software development. Therefore, it was not foolish for Symbolics to have the business plan it did.

Symbolics and LMI were founded by rivals at the MIT AI Lab, with Richard Greenblatt founding LMI and almost everyone else founding Symbolics, in particular Moon, Weinreb, Cannon, and Knight.

One of the factors in the adoption of Symbolics Lisp machines and Zetalisp was the fact that the first 3600's did not have a garbage collector, which meant that the performance penalty of garbage collecting a large address space was not observed. Originally the 3600 was to have a Baker-style incremental stop-and-copy collector, but because the address space was so large, ordinary programs did not exhaust memory for several days and intensive ones could run for about 8 hours. There was a facility for saving the running image, which basically did a stop-and-copy garbage collection to disk, and this image could be resumed. Therefore, instead of garbage collecting on-the-fly, a programmer would run until memory was exhausted, then he would start up the lengthy (up to several hours) process of disk-saving, and he would restart his program after dinner or the next day.

The incremental garbage collector was released several years after the first 3600's, and it proved to have relatively bad performance, possibly due to paging problems. Instead, Moon developed an ephemeral garbage collector that is similar to the Ungar generation scavenger collector developed for Smalltalk [?]. with generation scavenging, objects are promoted from one generation to the next by a stop-and-copy process. After several generations objects are promoted (tenured) to long-term storage. The idea is that an object will become garbage soon after its creation, so if you can look at the ages of objects and concentrate on only young objects, you will get most of the garbage and because a small working set is maintained paging performance is good.

Ephemeral garbage collection [Moon, 1984] is similar but maintains a few consing areas representing generations and a list of regions of memory where pointers to objects in the consing areas were created, and those regions are scanned in a stop-and-copy operation, moving from one generation to the other. Because objects in Smalltalk are created less frequently than in Lisp, the tradeoffs are a little different and the data structures are different.

The ephemeral garbage collector proved effective, and several years after the first 3600 was sold, an effective garbage collector was operational. It took a while for users to get used to the performance differences, but by then the 3600 had already established an acceptance group that was in the ascendancy, and the position of the 3600 was firmly implanted.

PERQ entered the market with a poorly performing microcoded machine that had been used as a document preparation computer. Scott Fahlman was involved with the company, and when Common Lisp made its debut, the Spice Lisp code was a nearly compliant Common Lisp, so PERQ was the first Common Lisp available on a Lisp machine.

The original PERQ and its later versions never made much of a commercial impact outside the Pittsburgh area, probably because its performance and price-performance were relatively poor.

We already saw the early history of the Xerox D-machines. Before the Common Lisp era, their

use became widespread in former InterLisp-10 circles.

Texas Instruments began to enter the Lisp machine business just before CLtL1 was published, in early 1984. They began with the Viking project, which was working with Spice Lisp and the MC68020. Later, they decided to go the pure Lisp machine route and introduced the Explorer, which was a microcoded machine that also ran the MIT software. Later, TI joined with LMI to trade some technology, which seemed to have little or no effect on the business fortunes of either company, except to inject capital into LMI, prolonging its existence. The Explorer had good price-performance, and also decent performance, and the high favor in which TI was held by the Department of Defense resulted in good sales for TI during the early Common Lisp era.

2.11.3 MacLisp on the Decline

Though still in widespread use, and spreading a bit, the development of MacLisp halted during the early 1980's. At this point, MacLisp ran on ITS, Multics, Tenex, TOPS-10, TOPS-20, and WAITs. Though only available on PDP-10's or related machines, these included the PDP-10, the DEC-10, the DEC-20, and the Foonly F2 and F4.

Funding for MacLisp development had been provided by the Macsyma Group, because the primary client for MacLisp, from the point of view of MIT, was the people who used Macsyma—the Macsyma Consortium. From the point of view of the rest of the world, however, Macsyma was an interesting application of Lisp, but MacLisp was of much wider appeal as a research and development tool for AI, particularly vision and robotics. However, these groups were not flush with funding, and, in any event, none found any reason to do other than to accept the use of a freely available MacLisp.

The funding for MacLisp was supplanted by funding for NIL on the Vax by the Department of Energy. The Department of Energy oversaw such things as research and development of nuclear weapons in addition to its more benign projects such as civilian energy. Therefore, the DOE was an alternative source of defense funding, and it funded such projects as S1 Lisp.

In general, each site that used MacLisp had a local wizard who was able to handle most of the problems encountered, possibly by consulting Jon L White. In at least one case, funding was made available to MIT to do some custom work. For instance, the single-segment version of MacLisp on WAITs was paid for by the Stanford AI Lab, and the work was done on site by Howard Cannon.

MacLisp was the host for a variety of language development and features over the years, including MicroPlanner, Conniver, Scheme, Flavors, Frames, Extends, Qlisp, and various vision-processing features. The last major piece of research in MacLisp was the multi-program programming environment done by Martin E. Frost and Gabriel at Stanford [?]. This environment defined a protocol that allowed MacLisp and E, the Stanford display editor which had operating-system support, to communicate over a mailbox-style operating system mechanism. With this mechanism, the code devoted to editing was shared by any users using E (even for non-Lisp tasks) by using the timesharing mechanism of the underlying host computer, and frequently code executed for the purpose of editing was executed within the operating system, requiring no code to be swapped in or paged in. It was also possible for Lisp programs to control the editor, so that very powerful “macros” written in Lisp could be used instead of the arcane E macro language. This environment predated similar Lisp/Emacs environments by a few years.

However, in the early days of the Common Lisp group, funding for NIL for the Vax by DOE and the Macsyma Consortium was halted, perhaps fueled by the belief that Lisp machines would run Macsyma well, perhaps fueled by the belief that the development of Common Lisp would provide the common base for Macsyma.

Around the same time that DOE funding stopped, Symbolics started a Macsyma Group to sell

Lisp-machine-based Macsyma. This group would remain profitable until its dissolution in the late 1980's or early 1990's.

With NIL funding stoppage, Jon L White joined Xerox to work on Interlisp—a stranger situation is difficult to imagine. First, White was apparently so clearly a Easterner in personality that the whole aura of the California lifestyle seems too foreign for him to accept. Second, the rivalry between MacLisp and InterLisp over the years would seem to obviate them working together.

2.12 Standards Development: 1984–1992

The period just after the release of *Common Lisp: The Language* (CLtL1) marked the beginning of an era of unprecedented Lisp popularity. In large part this popularity was coupled with the popularity of AI, but not entirely. Let's look at the ingredients: For the first time there was a commonly agreed standard for Lisp, albeit a flawed one; AI was on the rise, and Lisp was the language of AI; there appeared to be a burgeoning workstation market, and the performance of the workstations on Lisp was not far off from the Lisp machines; the venture capital community was looking at the success of companies like Sun, were awed by the prospects of AI, and had a lot of money as a result of the booming economy in the first half of the Reagan presidency; and computer scientists were turning into entrepreneurs in droves, spurred by the near-instant success of their colleagues at companies like Sun and Valid. Articles about Lisp were being written for popular magazines, requests for Common Lisp were streaming into places like CMU (Fahlman) and Stanford (Gabriel), and otherwise academic-only people were asked to speak at industry conferences and workshops on the topics of AI and Lisp and were regarded as sages of future trends.

The key impetus behind the interest by industry in Lisp and AI was that the problems of hardware seemed under control and the raging beast of software was next to be tamed. More traditional methods seemed inadequate, and there was always the feeling that the new thing, the radical thing would have a more thorough effect than the old, conservative thing—this is simply the allure of cachet, and it attracted both businessmen and venture capitalists.

2.12.1 Overview of Lisp Companies

As early as 1984—the year CLtL1 was published—several companies were founded to commercialize Common Lisp. These included Franz, Inc, Gold Hill, and Lucid, Inc. Other companies on the fringe of Lisp joined the Lisp bandwagon with Common Lisp or with Lisps that were on the road to Common Lisp. These included Three Rivers (PERQ) and TI. Some mainstream computer manufacturers joined in the Lisp business. These included DEC, HP, Sun, Apollo, Prime, and IBM. Some European companies joined the Common Lisp bandwagon, like Siemens and Honeywell Bull. And the old players began work on Common Lisp. These included the Lisp machine companies and Xerox. A new player from Japan—Kyoto Common Lisp (KCL)—provided a bit of a spoiler: KCL has a compiler that compiles to C, which is compiled by the C compiler. This Lisp was licensed essentially free, and the Common Lisp companies suddenly had a surprising competitor.

Though one might think a free, good-quality product would easily beat an expensive better-quality product, this proved false, and the Common Lisp companies thrived despite their no-cost competitor. It turned out that better performance, better quality, commitment by developers to moving ahead with the standard, and better service were more important than no price tag.

2.12.2 Common Lisp Companies

Franz, Inc., was already in business selling Franz Lisp, the MacLisp-like Lisp dialect used to transport a version of Macsyma called Vaxsyma. Franz Lisp was the most popular dialect of Lisp on the

Vax until plausible Common Lisps appeared. Franz decides to go into the Common Lisp market, funding the effort with the proceeds from its Franz Lisp sales. The principal founders of Franz are Fritz Kunze, John Foderaro, and Richard Fateman. Kunze is a PhD student of Fateman's at the University of California at Berkeley in the mathematics department. Kunze, who left the university to start Franz, proves to be a street-wise businessman who is able to rally his programmers in difficult times and to hang in when the Lisp business fades at the end of the decade. Foderaro is also a former student of Fateman's with a PhD in mathematics from Berkeley. He is the primary architect and implementor of the various Lisps offered by Franz, Inc. He is a quiet, thoughtful person in public, and though easily capable of making important contributions to the ongoing Common Lisp work, his public shyness prevents him from playing a dominant role. Fateman, one of the original implementors of Macsyma at MIT, carried the MacLisp/Lisp torch to Berkeley, and he was responsible for porting Macsyma to the Vax. There was a controversy about whether Macsyma, developed in part under DOE funding, was the property of MIT or in the public domain. Fateman contended it was in the public domain, and so he brought together a group to put a Lisp on the Vax suitable to support Macsyma—this was Franz Lisp. MIT was displeased at his actions.

Franz adopted a direct-sales strategy, in which the company targeted customers and sold directly to them.

Gold Hill was a division of a parent company named Apiary, Inc., which was founded by Carl Hewitt and his student Jerry Barber. Before founding Apiary/Gold Hill, Barber had spent the year at INRIA in France, where he wrote a MacLisp/Zetalisp-like Lisp for the IBM PC. This work was partly funded by INRIA. When he returned, his dialect was close enough to Common Lisp that Hewitt and Barber thought they could capitalize on the wave of Common interest by selling the existing Lisp as a Lisp about to become Common Lisp. Because the PC was believed to be an important machine for AI, it seemed to be an ironclad business plan, in which a variety of glamorous East-coast venture capitalists invested. Gold Hill's Lisp was not a Common Lisp and in the early years the company endured some criticism for false advertising, but as the Lisp was transformed into a Common Lisp, its quality apparently dropped. At the same time "AI winter" hit and Gold Hill was not able to survive at the level it once had. It was abandoned by its venture capitalists, laid off just about all its employees, and continues today as a two-man operation.

"AI winter" is the term first used in 1988 to describe the unfortunate commercial fate of AI. From the late 1970's and until the mid-1980's, artificial intelligence was an important part of the computer business—many companies were started with the then-abundant venture capital available for high-tech start-ups. By 1988 it became clear to business analysts that AI would not experience meteoric growth, and there was a backlash against AI and, with it, Lisp as a commercial concern. AI companies started to have substantial financial difficulties, and so did the Lisp companies.

The original Gold Hill Lisp was a dynamically scoped Lisp-machine-Lisp-like Lisp. The transition to Common Lisp took a number of years. During the period of switching to Common Lisp, Gold Hill tried to diversify, putting together an expert system shell. This shell was released at about the same time as the AI winter hit, and so it did not rescue the company as it was expected to.

Gold Hill sold direct as well.

Lucid, Inc. was founded by Gabriel (Stanford), Rod Brooks (MIT), Eric Benson (Utah/PSL), Scott Fahlman (CMU), and a few others. Backed by venture capital, Lucid adopted a different strategy than the other Common Lisp companies. First, it wrote an implementation of Common Lisp from scratch, and, second, it adopted an OEM strategy. The idea of the OEM route is to make arrangements with computer (hardware) company to market and sell Lisp under its own name. However, the Lisp is implemented and maintained by an outside company, in this case, Lucid, which would collect royalties. Lucid quickly struck OEM deals with Sun, Apollo, and Prime. The reason

this was possible was that Lucid traded on the strength of the names of its founders and the fact that it was writing a Common Lisp from scratch and would, therefore, be the first true Common Lisp.

Sun was signed partly because some of the founders of Sun were from Stanford and were familiar with Gabriel and his role in the definition of Common Lisp.

Eventually Lucid ported its Lisp and established OEM arrangements with IBM, DEC, and HP. Though the royalties were relatively small per copy, the OEM route established Lucid as the primary stock-hardware Lisp company. Because the hardware companies were enthusiastic about the business opportunities for AI and Lisp, they invested a lot in the business. Often they would pay a large porting fee, a fixed-price licensing fee, maintenance fees, as well as royalties. Getting Sun as an OEM was the key for Lucid's survival, because Sun workstations developed the cachet that was needed to attract customers to its Lisp. Sun was always regarded as leading-edge, and so people interested in leading-edge AI technology headed first to Sun. Sun also employed a number of engineers who did their own Lisp development, mostly in the area of programming environments.

Before AI winter hit, Lucid began diversifying into other languages (C/C++) and programming environments. In the latter part of the decade, Lucid began trying to change from an OEM company to a direct sales company, partly because the OEM companies were less interested in spending development and marketing dollars on Lisp when the return was not as high as it once was and certainly not as high a percentage of revenues as before.

2.12.3 Other Lisp Machine Companies

Three Rivers made a second Lisp machine, which had better performance than its previous model, but still not nearly as good as the 3600 or even as the faster MC68020-based Common Lisps. Though the Lisp was upgraded to be a fairly complete Common Lisp, this was just never a successful company. In the latter part of the decade it went out of business.

TI switched to being a Lisp machine company after the ill-fated Viking project. TI hired a couple of good people, including Patrick Dussud (who will appear later in the CLOS story). He developed a close working relationship with Dave Moon, of Symbolics, even though TI had a business arrangement with LMI. Through this relationship, both companies were able to improve the performance of its Lisp software, so that Symbolics and TI were the primary Lisp machine vendors by the third quarter of the decade.

Because of the arrangement with LMI, TI was able to develop its own microcode on LMI hardware before the Explorer was completed. TI had the right to use LMI software, but chose to use the MIT software instead. LMI had the right to remarket the Explorer, but chose not to.

TI made three models, the Explorer I, the Explorer II, and the Explorer II+. They also made an elaborate Lisp chip under contract to DARPA. The Explorer II and II+ both used this chip.

TI also made a board-level Lisp machine called the Micro Explorer. This machine also used the Lisp chip.

2.12.4 Big Companies With Their Own Lisps

DEC and HP implemented their own Lisps. DEC started with the Spice Lisp code and HP with PSL. Each company believed that AI would take off and that having a Lisp was an essential ingredient for success in the AI business. Both DEC and HP made arrangements with the original implementors of those Lisps, both by hiring students who had worked on them and by arranging for on-going consulting. Both DEC and HP grew fairly large businesses out of these Lisp groups, large by the standards of other Lisp companies. At the peak of Lisp in the last quarter of the

1980's, the main players in the Lisp business by revenue were Symbolics, TI, DEC, HP, Sun, and Lucid.

DEC and HP put a lot of effort into their Lisp offerings, primarily in the area of environments but also in the performance of the Lisp system itself.

In the last quarter of the 1980's, HP realized that PSL was not the winner and they needed to provide a Common Lisp. They chose Lucid to provide it, and they reduced their own engineering staff, choosing to focus more on marketing. Since the AI winter was just about upon them when they made the decision, it is not clear whether their perception of this situation forced them to cut back on their Lisp investment.

In the early 1990's, in the midst of AI winter, DEC also decided to abandon its own efforts and also chose Lucid.

2.12.5 IBM

IBM had a number of platforms suitable for Lisp: the PC, the mainframe, and the RT. Of these, IBM initially decided to put a Common Lisp only on the RT. IBM funded a pilot program to put Spice Lisp (Common Lisp) on the RT, which was to be IBM's first real entry into the workstation market. Fahlman, chief honcho of Spice Lisp, was also a founder of Lucid, and thanks to this connection a contract was eventually written for Lucid to port its Lisp to the RT for IBM. The RT did not have a native window system at the time the port had to be done, though later they did, and the contract required Lucid to deliver a complete environment. A lot of effort was put into providing this environment, along with the low-level window system on which the higher-level Lisp window system was written, but the RT was a commercial flop, primarily because it had such poor performance.

Later, IBM reentered the workstation market with its RS6000, which has good performance, and Lucid did the Common Lisp for it. IBM eventually contracted with Lucid to provide the same Lisp on the 370 and PS-2 running AIX, a version of Unix. Though it would seem to make sense that IBM would choose to work with just one company for its Lisp systems, the situation was that different business units were responsible for each hardware family mentioned, and there was no communication within IBM between any of them. Therefore, it was interesting and relevant that one business unit selected one particular vendor, but not compelling. The negotiations for each of these platforms was as if with different companies.

2.12.6 Xerox

Xerox produced a Common Lisp compatibility package on top of Interlisp. This package was never really a strong success for Xerox, who in the late 1980's got out of the Lisp business, licensing its Lisp software to a spinoff started by Xerox called Envos in 1989.

Envos put out a real Common Lisp implementation and sold the InterLisp-D environment, plus some other Xerox software such as Notecards and Rooms. Envos was funded by Xerox, but went out of business three years after it was founded. What was left of Envos became the company Venue, which essentially was granted the rights to continue marketing that same software, but without direct funding. The funding was provided by servicing Xerox's Lisp customer base (maintenance).

2.12.7 Kyoto Common Lisp

The last major player is Kyoto Common Lisp (KCL). This Lisp was developed at Kyoto University by the then-students Yuasa and Hagiya (???). These students were not part of the Common Lisp community, nor of the US Lisp community, and their implementation was based on a fresh reading

of CLtL1. Though they made several interesting misreadings of the book, the result was a very faithful Common Lisp.

Furthermore, they used an implementation technique not used in the US: they compiled Lisp code into C code, which was passed through the C compiler. KCL provided a runtime that supported garbage collection, IO, and the like, but porting this Lisp to a new machine took only a few days up to a few weeks, depending on how faithful the C compiler for that machine was to K&R C and how difficult the operating system dependencies were to port.

KCL was licensed for free in the US, and one US Common Lisp company—Ibuki in California—basically repackages it and supports it, though they provide some new development.

That Lisp, and Common Lisp, could be effectively implemented like this had never seriously occurred to any major US Lisp implementor. It was thought that compilation could not be as good because the compiler that did the low-level code generation would not have the original Lisp construct available, only an automatically generated sequence of C statements. Furthermore, compilation speed was thought need to suffer, because of double compilation. Though sluggish, this mode of compilation is not much worse than the generally sluggish compilers produced by Lucid, Franz, and the others.

So, not only did the implementation technique not have any serious drawbacks, but the porting benefits were significant. Nevertheless, KCL is, in the early 1990's, used primarily in Universities, where zero cost is important, and there is not much chance that KCL will be brought up to the language level of CLtL2 or ANSI Common Lisp, when it appears.

2.12.8 Obsession with Performance

In 1981, Larry Masinter and Dick Gabriel begin a study of how to evaluate the performance of a Lisp system. This exercise was done to address the variety of casual benchmarks that users had begun to do. Users are interested in such benchmarks because they want to find out which implementation—and perhaps which dialect—has superior performance. Sometimes benchmarkers wish to prove that one dialect is superior to others. But, because the Lisp language is so complex and Lisp implementation so involved, it is easy to end up comparing apples and oranges.

Among other things, Lisp treats at a uniform linguistic level both primitives and abstractions. For example, in most Lisp manuals, SETQ and APPEND are described in similar ways, so that it is reasonable to infer that SETQ and APPEND both take unit time, and usually just one machine instruction. However, APPEND is a complicated operation that involves finding the end of a list by linearly scanning it from the beginning. Users not familiar with Lisp implementation and representation will be easily confused.

Furthermore, there are almost always a variety of tradeoffs an implementor can make, trading space for time, time of one category of operation for time of another, worst-case time for average-case time for best-case time, and so on.

Masinter and Gabriel tried to explain all of the factors they could think of, and they presented this paper at the 1982 ACM Symposium on Lisp and Functional Programming [Gabriel, 1982].

Based on this paper, Gabriel obtained an ARPA contract to define and gather a series of benchmarks and to compare as many existing implementations as possible, along with an analysis of what the benchmarks showed. The work was completed in 1984, culminating in the publication of the results in a book; the so-called Gabriel benchmarks [Gabriel, 1985] emerge as the definition of Lisp performance, comparable in social function to Jack Dongarra's LINPACK measurements of supercomputer performance [?].

With the emergence of Common Lisp and commercial Lisp implementations, the users' obsession with performance found a convenient outlet in the Gabriel benchmarks—they provided an

apparently objective measure of performance, and so prospective customers insisted on seeing the results of the Gabriels.

At trade shows, some companies refused to show their Gabriel performance numbers for fear the competition would take advantage of them.

The situation became more complex and confrontational when Gabriel started a Lisp company, which used performance as a central sales weapon. And because the Gabriel benchmarks were developed by Gabriel, some regarded the tactic as unethical. Franz called the benchmarks the “Stanford Performance Evaluation Benchmarks” instead of the inflammatory “Gabriel Benchmarks.”

2.12.9 Japan

During this period Japan awakened to the use of Lisp, and some companies sponsored Lisp work despite the apparent endorsement of Prolog by MITI. In part this was spurred by Kyoto Common Lisp, which began to take hold in Japan. In part, also, this was spurred by former students who learned Lisp in the US as graduate students, students who became senior professors in major Japanese universities or who became senior managers in Japanese companies.

Of particular interest are the various blendings of Lisp with logic programming or object-oriented programming. The best example of this was Tao [Okuno, 1984], which is a blending of all three influences. A machine, called Elis, was designed and built by NTT (Nippon Telephone & Telegraph) to run Lisp and Tao. This machine never caught on in Japan or anywhere else.

Most of the large Japanese computer companies implemented Common Lisp either from scratch, from the base of another Lisp, or from KCL. By the time the ISO group started, there were several national standards groups in Japan interested in Lisp. One was JIS (Japan Information Systems), which corresponds to ANSI in the US. Language committees in JIS are dominated by senior professors, and the JIS Lisp committee—headed by Professor Takayasu (???) Ito, a student of McCarthy’s—was interested in Lisp from an academic point of view and did not take Common Lisp as seriously as JEIDA.

JEIDA is a loose association of companies, again unified by a professor, but in this case the professor is the young translator of CLtL1, Masayuki Ida. Ida is from a smaller, less prestigious university than Ito, who is a senior professor at one of the national universities, but Ida was the respect of industry groups because he is concerned with the commercial success of Lisp and those who use it. Ito, on the other hand, is concerned with elegant semantics suitable for building mathematical theories rather than the practical use of the language.

We will see more of these two later in the story of standardization.

2.12.10 DARPA and the SAIL Mailing Lists

Right after CLtL1 was published in 1984, ARPA was renamed DARPA, standing for Defense Advanced Research Projects Agency. Around this time, Steve Squires joined DARPA from NSA, and he brought a sort of enthusiasm and a high regard for politics not recently seen at the Office, as it was called. He was like the spider at the center of the web, his fingers feeling every twitch of a strand and urging his direction on all who touched on it. Seeing that Common Lisp could be legitimately called one of the shining successes of DARPA of that decade, he was quick to solidify what could be, based on the facts, a rather tenuous claim. That is, though DARPA played a minor role up to this point, he started actions to increase DARPA’s role and to try to press the advantage.

The first action was to organize a post-CLtL community meeting at the Naval Post-graduate School at Monterey, California. Though the Post-graduate School was one of the regular places DARPA held meetings, it was somewhat of an exotic location for the generally homebound Common

Lisp Group. And though there were natural leaders in the Common Lisp Group, DARPA chose some of its regular leaders to organize the sessions. The meeting was held in late 1984.

The main question regarded how to best extend the standard into areas that everyone knew was really part of Lisp but which were not heretofore standardized. These included: multi-programming, windows, error handling, object-oriented programming, subsets, graphics, foreign function calling, iteration, the yellow pages, the blue pages, and environments. The result of the meeting—besides confusion among the Common Lisp Group about what DARPA was up to—was a set of mailing lists, one for each area of interest. Since Gabriel administered the original lists—on the theory that the person who does the dog work eventually becomes top dog—he volunteered to maintain these lists. [?]

The following year saw little activity on these lists, because Lisp implementors, who make up the bulk of the participants, are busy implementing Common Lisp or trying to make their companies go.

2.12.11 The Start of X3J13

In a follow-up meeting one year later—December 1985 in Boston, Massachusetts—an apparently benign technical meeting was interrupted by a shocking announcement. The technical agenda regarded three main issues: object-oriented programming, windows, and the needs of users. The object-oriented programming part of the agenda was dominated by presentations on CommonLoops, Object Lisp, HP Object-oriented programming (HPOOPS, later called Common Objects), and New Flavors. The window part of the agenda was a presentation of Lucid’s RT-originated window system and Common Windows, a window system produced by Intellicorp. The user-concerns part of the meeting was just a discussion of why the users—AI companies—need more than Lisp, particularly object-oriented programming and windows.

The bombshell was that Steve Squires introduces Bob Mathis to talk about the need to establish an ANSI standards group for Common Lisp. The news heard was this: There was a plot afoot to start an ISO standards group for Lisp, and the only way that the US could respond was to have a Lisp standards group which sends a representative in the name of ANSI. There were persistent rumblings, the Group was told, that the Europeans do not like Common Lisp and wished to create a non-Common-Lisp standard. This effort was being lead by the French and British, and the US wanted to avoid the APL by letting the “frogs” get the convenorship.

This slip of the tongue was especially damaging, because sitting in the back of the room was Jerome Chailloux, the leader of the EuLisp effort. Sitting with him are Pierre Cointe, the leading French object-oriented programming guy, and several other Europeans.

Bob Mathis was sent by DARPA to save the day, by heading off the initial ISO thrust. He did this by recommending a study group that Mathis would head to determine whether Lisp was suitable for standardization. This group was to report in one year. This gave the US a year to start its standards committee and plot its strategy.

Mathis was the convenor for the ISO ADA working group, so he has experience shepherding standards through the process. He was asked by Squires to help the fledgling Common Lisp Group. Stunned, the Group was not sure how to respond, except to wonder at the severity—nay, the existence—of the threat. Nevertheless, it appears the wagon train is leaving, and all that was left to do at this meeting is to schedule the first organizational meeting for what would become X3J13, with the mysterious Bob Mathis at the helm.

A consummate politician, Mathis enlisted the aid of Fahlman, Steele, and Gabriel in rallying the troops.

DARPA’s stated reason for fearing the European effort was that there was a law that says that

if there is an ISO standard for a language, the US has to use it in defense-related work. Of course, there are many ways to circumvent this law with exceptions and such things, but possibly, the group was told, these mechanisms would not always work, and perhaps there may not be such a Common-Lisp-friendly set of program managers at DARPA.

The route chosen was the X3 route, rather than IEEE. X3 is a set of standards committees having to do with computers and information processing, all within the organization called CBEMA. Committees under X3 are authorized to submit proposed standards to ANSI, which then executes the public review process and publishes the standard. This arrangement is to prevent the technical experts from being sued by companies, as happened when some companies that used COBOL sued the individual technical experts for jeopardizing their companies by creating a COBOL standard that would cost a lot of money to adapt to. ANSI feared it would be hard to find technical experts to do the standards work if they could be sued as individuals for their efforts.

[The following text is very sketchy and needs to be filled out.]

- A few months later there is a meeting at SPARC in DC to start off the X3 activities. At this meeting the goals of standardization are discussed, and the biggest topic, pushed by DARPA in the names of Squires and Scherlis, is whether to merge with the Scheme activities—the technical issues surround macros and being a 1-Lisp. Also the goals of the new X3 group (to be X3J13) were discussed. Bobrow and Gabriel did a lot of the talking, but the nature of this meeting is somehow hard to remember.
- In a stupidly valiant effort, Gabriel tries to approach the Scheme community about merging, but they back away real fast. Clinger is one of the few who is sympathetic to the idea, but the combination of him and Gabriel is not enough. About this time, the Scheme community starts denying that Scheme is a dialect of Lisp, claiming closer bonds to Algol.
- Gabriel and Pitman, with the aid of Clinger and Sussman/Abelson, produce a report detailing the technical issues involved in macros and 1-lisps. There is a thesis by Kohlbecker [Kohlbecker, 1986b] that offers hope, and several alternatives are considered in the paper. This paper is published in LASC [Gabriel, 1988]. The idea for merging dies.
- In summer 1986, L&FP is held at MIT. At this meeting is the first time the European Lisp community and the US Lisp community square off. The EuLisp committee presents an unCommon Lisp paper along with a proposal for a layered Lisp definition [Padget, 1986]. There is a meeting in a downstairs room of the conference facility, where it is clear that neither group trusts the other. The US presents the view that they started X3J13 only because of fear that the Europeans would try to sink Common Lisp by starting an ISO committee, and the Europeans claim they are trying to start an ISO committee to stop X3J13 from sinking their Lisp efforts. Mathis has been trying to fight off an ISO group at that level, but it is about to come to a head.
- In November 1986 Gabriel attends a EuLisp meeting. EuLisp is most clearly a successor to LeLisp, but it is a European effort to define a layered language that will run on a variety of small to large machines, with object-oriented facilities, modules, and the like. It is a goal to be largely compatible with Common Lisp. Chailloux's position appears to be to try to build a bridge to Common Lisp. He attends several X3J13 meetings as an observer, and tries to work with Gabriel on forging a compromise.
- X3J13 starts some working groups to make some large proposals. These are: Iteration, compilation, editorial, and object-oriented programming. Kathy Chapman of DEC is named editor. DEC has picked the wrong time to provide an editor for 2 years.

- Four groups vie for defining the object-oriented programming part of Common Lisp: New Flavors (Symbolics) [Symbolics, 1985], CommonLoops (Xerox) [Bobrow, 1986], Object Lisp (LMI) [Drescher, 1987], and Common Objects [Kempf, 1987]. After a 6-month battle the following group is told to go write the standard for CLOS based on CommonLoops and a little New Flavors: Moon (Symbolics), Bobrow (Xerox), Kiczales (Xerox), Sonya Keene (Symbolics, a writer), Linda DeMichiel (Lucid), and Gabriel (Lucid). The last two end up writing the specification, but are in the group to represent a neutral influence. Also, there is a group of informed people who are informally in the group: Patrick Dussud (TI), Jim Kempf (Sun), and Jon L White (Lucid).
- ISO calls for a vote to start a work group (WG16), and X3J13 votes in favor. Through political maneuvering, France is the convenor and the US is the editor. In X3J13, Mathis is voted chairman, Steele vice chairman, Gabriel in international representative. Gabriel and Clinger are named the US editors for ISO Lisp. The first meeting is in Paris in February 1987. It is preceded by a little workshop on various Lisp topics. The US sends a large delegation, some on the nickel of private companies. DARPA pays for Mathis, Gabriel, and Clinger. Also there are McCarthy—to lend credence to the US seriousness—Masinter, Gregor Kiczales, and Kent Pitman. The US is given instructions to let nothing happen that jeopardizes Common Lisp in the US. A full day is taken up on the name (ISLISP) and goals. The goals include such things as clarity, conciseness, consistency, compactness. (??? Check these in the minutes). It is decided to start with Common Lisp, but to throw out various things. This is the French proposal. The US points out that this committee can only standardize existing things, while the Europeans seem happy to design something in the committee. Scheme is rejected as the basis, though the US casts 1/2 vote for Scheme and 1/2 vote for Common Lisp. Japan seems to vote with US, as does Canada. The Europeans are united. Christian Queinnec is the convenor, and he is a little weak. BSI tries to stop X3J13 through a technicality, but they are rejected.
- The early meetings of X3J13 concern goals and the like, and Mathis appears intent on preventing votes that change the language from taking place. Once the CLOS committee appears to be making progress, a cleanup committee is formed, and Larry Masinter (Xerox) is put in charge of it. Up to this point, Fahlman attended meetings, as does Clinger (representing Tektronics) and Chailloux (INRIA). Soon after the formation of the cleanup committee, these folks stop attending.
- Once there is a cleanup format for changes, there are a raft of changes made to Common Lisp. Hundreds of changes are made, many for no compelling reason. In particular, LOOP is adopted, the function type is cleaned up, compiler macros are adopted, CLOS is adopted (1988), a condition system is adopted, and lots, lots more. [Steele, 1990c] Worth discussing are some of these including conditions, iteration, CLOS, types, and the overbroad range of changes.
- In 1989 AI winter hits. The AI companies have oversold the capabilities of AI, and they pay a price. LMI folds around this time.
- Sun dominates the stock hardware market. They force Symbolics into a slump. In 1985 they had Sun 2's, which were inadequate, in 1986 they had Sun 3's, which were OK, and in 1988 they had Sun 4's, which were beginning to be pretty good. The IBM RT is a huge flop. HP tries to move to Common Lisp based on PSL, but gives up and chooses Lucid instead. It is beginning to be clear that Common Lisp, as predicted by Brooks and Gabriel, cannot hack it is a delivery solution.

- In 1989, the first non-intrusive garbage collectors on stock hardware appear (Lucid and Franz) [Sobalvarro, 1988].
- In 1990 TI drops out of the Lisp game. Xerox drops Interlisp as a commercial venture in 1989. IBM drops its Lisp370-based CL and adopts Lucid. In 1990 DEC drops its Lisp and adopts Lucid. It is clear that the AI winter and inability of the commercial Lisps to solve integration and delivery problems. Various approaches are tried (Treeshaking in Lucid and selective loading in Franz). These are failures. In 1990, Gabriel attempts to analyze the failure in a keynote address at Eurolisp [?; ?].
- In 1988 DARPA calls a meeting of Gabriel, Bobrow, Sussman, Fahlman, and maybe a few others at DARPA (Rosslyn) to discuss merging Scheme and Common Lisp again. Scherlis and Squires drive this. Another failure.
- Scheme begins IEEE standardization in 1989 and finishes in 1991. The first meeting is in the early winter of 1989, and Gabriel attends. He argues against standardization or joining X3J13 instead, but Scheme loathing of Common Lisp is too strong. There will be two tracks for Scheme: IEEE standard and informal Rⁿ Reports.
- Scheme, hygienic macros appear to make it into the Rⁿ Scheme.
- In 1990, the EuLisp group produces a draft of EuLisp. This is a layered language that has this: it is a 1-lisp, has modules, CLOS-like OOP layered in three nice layers, and a variety of cleanups of Common Lisp mistakes. This work is driven by Padgett, who primarily writes the spec.
- At ISO, by 1990, Japan has become a stronger voice, and the process is stalled. Gabriel resigns as head of the US delegation. Clinger had resigned in 1988. Kiczales is the acting representative.
- The last meeting of X3J13 to make decisions is supposed to be January 1989, but fervor to change things drags on. This is the “everyone must piss on it” syndrome.
- In 1990 Pitman volunteers to be the editor taking over for Chapman, who disappeared in 1989. In 1991 he produces a first draft that is 1300 pages long. One of the remaining Lisp companies (Ibuki) tries to get X3J13 to back down, hoping to revert back to CLtL1.

[End of sketchy section.]

analysis

[The following text is very sketchy and needs to be filled out.]

Market forces dominate acceptance groups. The acceptance group for CL goes on the decline in 1989, and a flurry of diversification takes place in terms of changes to CL, additions to CL, EuLisp finally emerging (Europe is hit with AI winter first). Some parts of the acceptance group heads for Scheme because of size and simplicity (e.g. extension language folks like CFI).

Consolidation of CL causes Interlisp to die. Cachet of Lisp machines first and Suns later pushes CL ahead. The post-AI-winter era has shrunk things down to small Lisp groups again, so we expect a period of diversification.

AI companies blame AI winter on the Lisp groups who did not provide delivery solutions, but the AI companies never communicated their needs to the Lisp companies, because of fear of collaboration (a common but stupid phenomenon that happens when academics become business people).

CLOS is a consolidation itself, and it has added a cachet to an otherwise stagnant Common Lisp.

Environments for Lisp never really were standardized, though the AI companies largely provided an environment plus a big library of UI and AI stuff.

[End of sketchy section.]

2.13 Expressiveness

Lisp has proved itself more concerned with expressiveness than anything else. We can see this by observing that only a person well-versed with how a particular Lisp is implemented can write efficient programs. Here is a perfectly nice piece of code:

```
(defun make-matrix (n m)
  (let ((matrix ()))
    (dotimes (i n matrix)
      (push (make-list m) matrix))))

(defun add-matrix (m1 m2)
  (let ((l1 (length m1))
        (l2 (length m2)))
    (let ((matrix (make-matrix l1 l2)))
      (dotimes (i l1 matrix)
        (dotimes (j l2)
          (setf (nth i (nth j matrix))
                (+ (nth i (nth j m1))
                   (nth i (nth j m2))))))))))
```

The expression to read and write a cell in a matrix looks perfectly harmless and fast as anything. But it is slow, because `nth` takes time proportional to the value of its first argument, essentially CDRing down the list every time it is called. (An experienced Lisp coder would iterate over the cells of a list rather than over numeric indices, or would use arrays instead of lists.)

Here expressiveness has gone awry. People tend to expect that operations in the language cost a small unit time, not something complicated to figure out. So, precisely *because* Lisp is so expressive, it is very hard to write fast programs, though it is easy to write pretty ones.

3 Evolution of Some Specific Language Features

In this section we discuss the evolution of some language features that are either unique to Lisp or uniquely handled by Lisp.

3.1 The Treatment of NIL (and T)

Almost since the beginning, Lisp has used the symbol `nil` as the distinguished object that indicates the end of a list (and which is therefore itself the empty list); this same object also serves as the *false* value returned by predicates. McCarthy has commented that these decisions were made “rather lightheartedly” and “later proved unfortunate.” Furthermore, the earliest implementations established a tradition of using the zero address as the representation of NIL; McCarthy also commented that “besides encouraging pornographic programming, giving a special interpretation to the address 0 has caused difficulties in all subsequent implementations” [McCarthy, 1978].

The advantage of using address 0 as the representation of `nil` is that most machines have a “jump if zero” instruction or the equivalent, allowing a quick and compact test for the end of a list.

As an example of the implementation difficulties, however, consider the PDP-10 architecture, which had 16 registers, or “accumulators”, which were also addressable as memory locations. Memory location 0 was therefore register 0. Because address 0 was `nil`, the standard representation for symbols dictated that the right half of register 0 contain the property list for the symbol `nil` (and the left half contained the address of other information, such as the character string for the name “`nil`”). The implementation tradition thus resulted in tying up a register in an architecture where registers were a scarce resource.

Later, when MacLisp adopted from Interlisp the convention that `(car nil) = (cdr nil) = nil`, register 0 was still reserved; its two halves contained the value 0 so that the `car` and `cdr` operations need not special-case `nil`. But all operations on symbols had to special-case `nil`, for it no longer had the same representation as other symbols. This led to some difficulties for Guy Steele, who had to find every place in the assembly-language kernel of MacLisp where this mattered. Nowadays he concedes that it was actually rather a good idea. The introduction of an abstraction for reliably obtaining the property list settled his last objections (item 4 of February 7, 1975):

The new SUBR of one argument, `PLIST`, is used to retrieve the property list of a symbol; this function works on all symbols including `NIL`, whereas `CDR` will work only for non-null symbols in non-`*RSET` mode [that is, with error checking turned off!]. Similar remarks hold for the new SUBR of two arguments, `SETPLIST`. (`SETPLIST X (PLIST X)`) should be a no-op.

Nowadays some Common Lisp implementations use a complex system of offset data representations to avoid special cases for either conses or symbols; *every* symbol is represented in such a way that the data for the symbol does not begin at the memory word addressed by a symbol pointer, but two words after the word addressed. A cons cell consists of the addressed word (the `cdr`) and the word after that (the `car`). In this way the same pointer serves for both `nil` the symbol and `()` the empty list pseudo-cons whose `car` and `cdr` are both `nil`.

There is a danger in using a quick test for the end of a list; a list might turn out to be improper, that is, ending in an object that is neither the empty list nor a cons cell. Interlisp split the difference, giving the programmer a choice of speed or safety [Teitelman, 1978]:

Although most lists terminate in `NIL`, the occasional list that ends in an atom, e.g., `(A B . C)`, or worse, a number or string, could cause bizarre effects. Accordingly, we have made the following implementation decision:

All functions that iterate through a list, e.g., `member`, `length`, `mapc`, etc., terminate by an `nlistp` check, rather than the conventional null-check, as a safety precaution against encountering data types which might cause infinite `cdr` loops ... [their italics]

For users with an application requiring extreme efficiency, [their footnote: A `NIL` check can be executed in only one instruction; an `nlistp` on Interlisp-10 requires about 8, although both generate only one word of code.] we have provided fast versions of `memb`, `last`, `nth`, `assoc`, and `length`, which compile open and terminate on `NIL` checks ...

Fischer Black commented as early as 1964 on the difference between `NIL` and `()` as a matter of programming style [Black, 1985]. There has been quite a bit of discussion over the years of whether to tease apart the three roles of the empty list, the false value, and the otherwise uninteresting symbol whose name is “`nil`”; such discussion was particularly intense in the Scheme community, many of whose constituents are interested in elegance and clarity. They regard such constructions as

```
(if (car x) (+ (car x) 1))
```

as bad puns, preferring the more explicit

```
(if (not (null (car x))) (+ (car x) 1))
```

The *Revised Revised Report on Scheme* [Clinger, 1985b] defined three distinct quantities `nil` (just another symbol); `()`, the empty list; and `#!false`, the boolean false value (along with `#!true`, the boolean true value). However, in an interesting compromise, all places in the language that tested for true/false values regarded both `()` and `#!false` as false and all other objects as true. The report comments:

The empty list counts as false for historical reasons only, and programs should not rely on this because future versions of Scheme will probably do away with this nonsense.

Programmers accustomed to other dialects of Lisp should beware that Scheme has already done away with the nonsense that identifies the empty list with the symbol `nil`.

The *Revised³ Report on the Algorithmic Language Scheme* [Rees, 1986] shortened `#!false` and `#!true` to `#f` and `#t`, and made a remark that is similar but more refined (in both senses):

The empty list counts as false for compatibility with existing programs and implementations that assume this to be the case.

Programmers accustomed to other dialects of Lisp should beware that Scheme distinguishes false and the empty list from the symbol `nil`.

The recently approved IEEE standard for Scheme specifies that `#f` and `#t` are the standard false and true values, and that all values except `#f` count as true, “including `#t`, the empty list, symbols, numbers, strings, vectors, and procedures” [IEEE, 1991]. So the Scheme community has, indeed, overcome long tradition and completely separated the three notions of the false value, the empty list, and the symbol `nil`. Nevertheless the question continues to be debated.

The question of `nil` was also debated in the design of Common Lisp, and at least one of the directly contributing implementations, NIL, had already made the decision that the empty list `()` would not be the same as the symbol `nil`. (A running joke was that NIL (New Implementation of Lisp) unburdened NIL of its role as the empty list so that it would be free to serve as the name of the language!) Eventually the desire to be compatible with the past, however cruffy [Raymond, 1991], carried the day. ; this was best expressed by the following piece of E-mail:

Date: Sunday, 14 March 1982 14:40-EST
 From: Symbolics Technical Staff
 Reply-to: Moon@SCRC-TENEX@MIT-MC
 To: Common-Lisp at SU-AI
 Subject: The T and NIL issues

I’m sorry this message has been so long delayed; my time has been completely occupied with other projects recently.

We have had some internal discussions about the T and NIL issues. If we were designing a completely new language, we would certainly rethink these, as well as the many other warts (or beauty marks) in Lisp. (We might not necessarily change them, but we would certainly rethink them.) However, the advantages to be gained by changing T and NIL now are quite small compared to the costs of conversion. The only resolution to these issues that Symbolics can accept is to retain the status quo.

To summarize the status quo: NIL is a symbol, the empty list, and the distinguished “false” value. SYMBOLP, ATOM, and LISTP are true of it; CONSP is not. CAR, CDR, and EVAL of NIL are NIL. NIL may not be used as a function, nor as a variable. NIL has a property list. T is a symbol and the default “true” value used by predicates that

are not semi-predicates (i.e., that don't return "meaningful" values when they are true.) EVAL of T is T. T may not be used as a variable. T is a keyword recognized by certain functions, such as FORMAT.

The behavior of LISTP is a change to the status quo which we [Symbolics] agreed to long ago, and would have implemented long ago if we weren't waiting for Common Lisp before making any incompatible changes. The status quo is that NIL has a property list; however, this point is probably open to negotiation if anyone feels strongly that the property-list functions should error when given NIL.

The use of T as a syntactic keyword in CASEQ and SELECTQ should not be carried over into their Common Lisp replacement, CASE. It is based on a misunderstanding of the convention about T in COND and certainly adds nothing to the understandability of the language.

T and NIL are just like the hundreds of other reserved words in Lisp, dozens of which are reserved as variables, most of the rest as functions. Any particular program that wants to use these names for ordinary symbols rather than the special reserved ones can easily do so through the use of packages. There should be a package option in the portable package system by which the reserved NIL can be made to print as "()" rather than "GLOBAL:NIL" when desired.

It is worth noting that Lisp implementors have not been tempted to identify `nil` with the *number* 0 (as opposed to the internal *address* 0), with one notable exception, a Lisp system for the PDP-11 written in the 1970's by Richard M. Stallman, in which the number 0 rather than the symbol `nil` was used as the empty list and as false. Compare this to the use of 0 and 1 as false and true in APL [Iverson, 1962], or the use of 0 as false and as the null pointer in C [Kernighan, 1978]. Both of these languages have provoked the same kinds of comments about puns and bad programming practice that McCarthy made about Lisp.

This may seem to the reader to be a great deal of discussion to expend in this paper on such a small point of language design. However, the space taken here reflects accurately the proportion of time and energy in debate actually expended on this point by the Lisp community over the years. It is a debate about expressiveness versus cleanliness and about different notions of clarity. Even if Lisp does not enforce strong typing, some programmers prefer to maintain a type discipline in their code, writing `(if (not (null (car x))) ...)` instead of `(if (car x) ...)`. Others contend that such excess clutter detracts from clarity rather than improving it.

3.2 Iteration

While Lisp, according to its P.R., has traditionally used conditionals and recursively defined functions as the principal means of expressing control structure, there have in fact been repeated and continuing attempts to introduce various syntactic devices to make iteration more convenient. This may have been driven by the desire to emulate styles of programming found in ALGOL-like languages and by the fact that, although some compilers would sometimes optimize tail-recursive calls, the programmer could not rely on this until the era of good Scheme and Common Lisp compilers in the 1980's, so performance was an issue.

Perhaps the simplest special iteration construct is exemplified by the first `do` loop introduced into MacLisp (in March, 1969):

```
(do var init step test . body)
```

means the same as

```
(let ((var init))
  (block (when test (return))
    (progn . body)
    (setq var step)))
```

Thus the Fortran DO loop

```
DO 10 J=1,100
  IF (A(J) .GT. 0) SUM = SUM + A(J)
10 CONTINUE
```

could be expressed as

```
(DO J 1 (+ J 1) (> J 100)
  (WHEN (PLUSP (A J))
    (SETQ TOTAL (+ TOTAL (AREF A J)))))
```

(except, of course, that Lisp arrays are usually 0-origin instead of 1-origin, so

```
(DO J 0 (+ J 1) (= J 100)
  (WHEN (PLUSP (A J))
    (SETQ TOTAL (+ TOTAL (AREF A J)))))
```

is actually a more idiomatic rendering).

This “old-style” MacLisp `do` loop was by no means the earliest iteration syntax introduced to Lisp; we mention it first only because it is the simplest. The Interlisp CLISP iterative statements were the earliest examples of the more typical style that has been reinvented ever since:

```
(FOR J←0 TO 99 SUM (A J) WHEN (PLUSP (A J)))
```

This returns the sum as its value rather than accumulating it into the variable `TOTAL` by side effect. (See section 3.7.1 for further discussion of Algol-style syntax.)

Macros or other code-transformation facilities such as CLISP make this kind of extension particularly easy. While the effort in Interlisp was centralized, in other Lisp dialects (MacLisp not excepted) there have been repeated instances of one local wizard or another cobbling up some fancy syntax for iterative processes in Lisp. Usually it is characterized by the kind of pseudo-English keywords found in the Algol-like languages, although one version at Stanford relied more on the extended-ASCII character set available on its home-grown keyboards. This led to a proliferation of closely related syntaxes, typically led off by the keyword `FOR` or `LOOP`, that have attracted many programmers but turned the stomachs of others as features accreted.

Because of these strong and differing aesthetic reactions to iteration syntax, the question of whether to include a `loop` macro became a major political battle in the early design of Common Lisp, with the Lisp Machine crowd generally in favor of its adoption and Scott Fahlman adamantly opposing it, seconded perhaps more weakly by Guy Steele. The result was a compromise. The first definition of Common Lisp [Steele, 1984] included a `loop` macro with absolutely minimal functionality: it permitted no special keywords, good only for expressing endless repetition of a sequence of subforms. It was understood to be a placeholder, reserving the name `loop` for possible extension to some full-blown iteration syntax. ANSI committee X3J13 did eventually agree upon and adopt a slightly cleaned-up version of `loop` [Steele, 1990c] based on the one used at MIT and on Lisp Machines (which was not very much different from the one in Interlisp).

In the process X3J13 also considered two other approaches to iteration that had cropped up in the meantime: `series` (put forward by Richard Waters) and `generators` and `gatherers` (by Pavel

Curtis and Crispin Perdue) [Steele, 1990c; Waters, 1984; Waters, 1989a; Waters, 1989b]. The example Fortran `DO` loop shown above would be rendered using series as

```
(collect-sum (choose-if #'plusp
                        (#M(lambda (j) (a j))
                          (scan-range :start 0 :below 100))))
```

This is a functional style, reminiscent of APL:

```
+/(T>0)/T←A[1:100]
```

The definition of the series primitives and their permitted compositions is cleverly constrained so that they can always be compiled into efficient iterative code without the need for unboundedly large intermediate data structures at run time. Generators and gatherers are a method of encapsulating series (at the cost of run-time state) so that they look like input or output streams, respectively, that serve as sources or sinks of successive values by side effect. Thus

```
(generator (choose-if #'plusp
                      (#M(lambda (j) (a j))
                        (scan-range :start 0 :below 100))))
```

produces an object that delivers successive positive numbers resulting from $(A\ 0)$, $(A\ 1)$, etc., when given to the extraction function `next-in`. This reminds one of the possibilities lists of Conniver [McDermott, 1974] or of the generators of Alphard [Shaw, 1981], though we know of no direct connection.

After much debate, X3J13 applauded the development of series and generators but rejected them for standardization purposes, preferring to subject them first to the test of time.

One other iteration construct that deserves discussion is the MacLisp “new-style” `do` loop, introduced in March, 1972:

```
(do ((var1 init1 step1)
     (var2 init2 step2)
     ...
     (varn initn stepn))
    (test . result)
    . body)
```

This evaluates all the *init* forms and binds the corresponding variables *var* to the resulting values. It then iterates the following sequence: if evaluating the *test* form produces a true value, evaluate the *result* forms and return the value of the last one; otherwise execute the *body* forms in sequence and repeat.

The beauty of this construct is that it allows the initialization and stepping of multiple variables without use of kitschy pseudo-English keywords. The awful part is that it uses multiple levels of parentheses as delimiters and you have to get them right or endure strange behavior; only a diehard Lisper could love such a syntax.

Arguments over syntax aside, there is something to be said for recognizing that a loop that steps only one variable is pretty useless, in *any* programming language. It is almost always the case that one variable is used to generate successive values while another is used to accumulate a result. If the loop syntax steps only the generating variable, then the accumulating variable must be stepped “manually” by using assignment statements (as in the Fortran example) or some other side effect. The multiple-variable `do` loop reflects an essential symmetry between generation and accumulation, allowing iteration to be expressed without explicit side effects:

```
(define (factorial n)
  (do ((j n (- j 1))
      (f 1 (* j f)))
      ((= j 0) f)))
```

It is indeed not unusual for a *do* loop of this form to have an empty body, performing all its real work in the *step* forms.

While there is a pretty obvious translation of this *do* construct in terms of *prog* and *setq*, there is also a perspicuous model free of side effects:

```
(labels ((the-loop
          (lambda (var1 var2 ... varn)
            (cond (test . result)
                  (t (progn . body)
                     (the-loop step1 step2 ... stepn))))))
  (the-loop init1 init2 ... initn))
```

Indeed, this is equivalent to the definition of *do* adopted by Scheme [IEEE, 1991], which resolves an outstanding ambiguity by requiring that the variables be updated by binding rather than by side effect. Thus the entire iteration process is free of side effects. With the advent of good Scheme compilers such as ORBIT [Kranz, 1986] and good Common Lisp compilers, compiling the result of this side-effect-free translation produces exactly the same efficient machine-language code one would expect from the *PROG*-and-*SETQ* model.

3.3 Macros

Macros appear to have been introduced into Lisp by Timothy P. Hart in 1963 in a short MIT AI Memo [Hart, 1963], which we quote here in its entirety [request for permission is pending]:

In LISP 1.5 special forms are used for three logically separate purposes: a) to reach the alist, b) to allow functions to have an indefinite number of arguments, and c) to keep arguments from being evaluated.

New LISP interpreters can easily satisfy need (a) by making the alist a *SPECIAL*-type or *APVAL*-type entity. Uses (b) and (c) can be replaced by incorporating a *MACRO* instruction expander in *define*. I am proposing such an expander.

1. The property list of a macro definition will have the indicator *MACRO* followed by a function of one argument, a form beginning with the macro's name, and whose value will replace the original form in all function definitions.
2. The function *macro*[1] will define macros just as *define*[1] defines functions.
3. *define* will be modified to make macro expansions.

Examples:

1. The existing *FEXPR* *csetq* may be replaced by the macro definition:

```
MACRO ((
  (CSETQ (LAMBDA (FORM) (LIST (QUOTE CSET) (LIST (QUOTE QUOTE)
    (CADR FORM)) (CADDR FORM))))
  ))
```

2. A new macro *stash* will generate the form found frequently in *PROG*'s:

```
x := cons[form;x]
```

Using the macro **stash**, one might write instead of the above:

```
(STASH FORM X)
```

Stash may be defined by:

```
MACRO ((
  (STASH (LAMBDA (FORM) (LIST (QUOTE SETQ) (CADAR FORM)
    (LIST (CONS (CADR FORM) (CADAR FORM)))) ))
))
```

3. New macros may be defined in terms of old. **Enter** is a macro for adding a new entry to a table (dotted pairs) stored as the value of a program variable.

```
enter[form]  $\stackrel{\text{MACRO}}{=}$  list[STASH;list[CONS;cadr[form];
caddr[form];caddr[form]]
```

Incidentally, use of macros will alleviate the present difficulty resulting from the 90 LISP compiler's only knowing about those fexprs in existence at its birth.

The macro defining function **macro** is easily defined:

```
macro[l]  $\equiv$  deflist[l;MACRO]
```

The new **define** is a little harder:

```
define[l]  $\equiv$  deflist[mdef[l];EXPR]
mdef[l]  $\equiv$  [
  atom[l]  $\rightarrow$  1;
  eq[car[l];QUOTE]  $\rightarrow$  1;
  member[car[l];(LAMBDA LABEL PROG)]  $\rightarrow$ 
    cons[car[l];cons[cadr[l];mdef[caddr[l]]]];
  get[car[l];MACRO]  $\rightarrow$  mdef[get[car[l];MACRO][l]];
  T  $\rightarrow$  maplist[l; $\lambda$ [[j];mdef[car[j]]]]]
```

4. The macro for **select** illustrates the use of macros as a means of allowing functions of an arbitrary number of arguments:

```
select[form]  $\stackrel{\text{MACRO}}{=}$   $\lambda$ [[g];
  list[list[LAMBDA;list[g];cons[COND;
    maplist[caddr[form]; $\lambda$ [[l];
      [null[cdr[l]]  $\rightarrow$  list[T;car[l]];
      T  $\rightarrow$  list[list[EQ;g;caar[l]];cadar[l]]]]]]
  ]];caddr[form]]][gensym[]]
```

There are a number of points worth noting about Hart's proposal. It allows the macro expansion to be computed by an arbitrary user-defined function, rather than relying on substitution into a template, as so many other macro processors of the day did. He noted that macros, unlike fexprs, require no special knowledge for each one on the part of the compiler. Macros are expanded at function definition time, rather than on the fly as a function is interpreted or compiled. Note the switching off between S-expression and M-expression syntax. The **STASH** macro is the equivalent of the **PUSH** macro found in Interlisp [Teitelman, 1978] and later in Common Lisp by way of Lisp-Machine Lisp; the verb "stash" was commonly used in the 1960's. There are two minor bugs in the

definition of `mdef`: first, `PROG` is not properly handled, because it fails to process all the statements in a `PROG` form; second, `COND` is not handled specially, which can lead to mistakes if a variable has the same name as a macro and the variable is used as the test part of a `COND` clause. (Perhaps this was an oversight, or perhaps it never occurred to Hart that anyone would have the bad taste to use a name for two such different purposes.) The last example illustrates the technique of generating a new name for a temporary binding to avoid multiple evaluations of an argument form. Finally, Hart achieved an amazing increase in expressive power with a deceptively simple change to the language, by encouraging the user to exploit the power of Lisp to serve as its own metalanguage.

Hart's macro language was subsequently used in the Lisp system for the Q-32 [Saunders, 1985b]. Inspection of the `MDEF` function in the compiler code [Saunders, 1985a, p. 311] reveals that the error in processing `PROG` statements had been repaired: `mdef[caddr[1]]` was replaced by `mdef[cddr[1]]`. (In fact, this may be what Hart had originally intended; in [Hart, 1963] the "a" appears to have been written in by hand as a correction over another letter. Perhaps the typist had made the typographical error `mdef[ccddr[1]]` and subsequently a wrong correction was made.) Unfortunately, the use of `mdef[cddr[1]]` has its own problems: a variable whose value is returned by a `LAMBDA` expression or a tag at the head of a `PROG` might be incorrectly recognized as the name of a macro, thereby treating the body of the `LAMBDA` or `PROG` form as a macro call. Picky, picky—but nowadays we do try to be careful about that sort of thing.

A similar sort of computed macro appeared in the MIT PDP-6 Lisp, but macros calls were expanded on the fly as they were encountered by the compiler or the interpreter. In the case of the interpreter, if an explicitly named function in a function call form turned out to have a `MACRO` property on its property list (rather than one of the function indicators `EXPR`, `SUBR`, `LSUBR`, `FEXPR`, or `FSUBR`) then the function definition was given the original macro call form as an argument and was expected to return another form to be evaluated in place of the call. The example given in the PDP-6 Lisp memo [PDP-6 Lisp, 1967] was

```
(DEFPROP CONSCONS
  (LAMBDA (A)
    (COND ((NULL (CDDR A)) (CADR A))
          ((LIST (QUOTE CONS)
                  (CADR A)
                  (CONS (CAR A)
                        (CDDR A))))))
  MACRO)
```

This defined a macro equivalent in effect to the Common Lisp function `list*`. Note the use of `DEFPROP` to define a function, the use of `QUOTE` rather than a single quote character, and the omission of the now almost universally customary `T` in the second `COND` clause.

An advantage of this scheme was that one could define some functions that use macros and later define (or redefine) the macro; a macro call would always use the latest macro definition. A drawback, however, was that the interpreter must constantly re-expand the same macro call every time it is repeated, reducing speed of execution. A device called *displacing macros* soon became common among MacLisp users; it involved the utility function `DISPLACE`:

```
(DEFUN DISPLACE (OLD NEW)
  (RPLACA OLD (CAR NEW))
  (RPLACD OLD (CDR NEW))
  OLD)
```

One would then write

```
(DEFPROP CONSCONS
  (LAMBDA (A)
    (DISPLACE A
      (COND ...)))
  MACRO)
```

The effect was to destructively alter the original list structure of the macro call so as to replace it with its expansion.

This all-too-clever trick had drawbacks of its own. First, it failed if a macro needed to expand to an atom (such as a number or variable reference); macro writers learned to produce `(PROGN FOO)` instead of `FOO`. Second, if a macro were redefined after a call to it had been displaced, subsequent executions of the call would not use the new definition. Third, the code was modified; pretty-printing code that originally contained a macro call would display the expansion, not the original macro call. This last drawback was tolerable only because the MacLisp environment was so firmly file-based: displacing macros modified only the in-core copy of a program; it did not affect the master definition, which was considered to be the text in some file. Displacing macros of this kind would have been intolerable in Interlisp.

Around 1978, Lisp Machine Lisp introduced an improvement to the displacement technique:

```
(defun displace (old new)
  (rplacd new (list (cons (car old) (cdr old)) new))
  (rplaca old 'si:displaced)
  new)

(defmacro si:displaced (old new) new)
```

The idea is that the macro call is displaced by a list (`si:displaced macro-call expansion`). The macro `si:displaced` transparently returns its second argument form, so everything behaves as if the expansion itself had replaced the macro call. While expanding the call to `si:displaced` is not free, it is presumably cheaper than continually re-expanding the original macro call (if not, then the macro writer shouldn't use `displace`). The Lisp Machine pretty-printer recognizes calls to `si:displace` and prints only the original macro call.

Returning to the 1960's: The `DEFPROP` notation was soon replaced by the `DEFUN` notation in MacLisp [White, 1969–1982, Jon L White, item 1 of March 1, 1969]:

`DEFUN` is an `FSUBR` used to define functions. Examples are

```
(DEFUN ONECONS (X) (CONS 1 X))
```

which is equivalent to

```
(DEFPROP ONECONS
  (LAMBDA (X) (CONS 1 X))
  EXPR)
```

and

```
(DEFUN SMASH FEXPR (L) (RPLACD L NIL))
```

[which] is equivalent to

```
(DEFPROP SMASH
  (LAMBDA (L) (RPLACD L NIL))
  FEXPR)
```

The novel feature of DEFUN is that one need not be so concerned with balancing parentheses at the very end of the function definition, since the type flag may be omitted if it is `EXPR`, and appears near the front of the DEFUN list if it is some other. Also, the `LAMBDA` need not be directly inserted.

White elsewhere further noted [?]:

[DEFPROP] has been the usual method of defining functions in MacLisp, but DEFUN has been implemented to provide a more clear indication that such a definition is being performed, and to allow experimentation in the structure of p-list and function representation.

DEFUN itself might easily have been implemented as a macro, but it was not; because it was used so frequently, for reasons of efficiency (to avoid consing) it was hand-coded as an `FSUBR` and recognized specially by the compiler. (Common Lisp eventually specified that `defun` should be a macro.)

BBN Lisp [Teitelman, 1971] had *three* kinds of macro: *open*, *computed*, and *substitution* (described below). A macro definition was stored in the property list of its name under the `MACRO` property; the form of the property value determined which of three types of macro it was. Originally all three types were effective only in compiled code. Eventually, however, after BBN Lisp became Interlisp [Teitelman, 1978], a DWIM hack called MACROTRAN was added that made all three types of macro effective in interpreted code. If interpreting a function call resulted in an “undefined function” error, the DWIM system would step in. MACROTRAN would gain control, expand the macro, and evaluate the resulting expansion. The Interlisp manual duly notes that interpreted macros will work only if DWIM is enabled. Contrast this with the MIT approach of building macros directly into the interpreter (as well as the compiler) as a primitive language feature.

A BBN-Lisp *open* macro simply caused the macro name to be replaced by a lambda expression, causing the function to be compiled “open” or in-line. Here is an open macro definition for `ABS`:

```
(LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X))))
```

Of course this has exactly the same form as a function definition.

A BBN-Lisp *computed* macro was similar to the kind in MIT PDP-6 Lisp, except that the expander function received the `CDR` of the macro call rather than the entire macro call. Here is a computed macro for `LIST`:

```
(X (LIST (QUOTE CONS)
         (CAR X)
         (AND (CDR X)
              (CONS (QUOTE LIST)
                    (CDR X)]
```

The leading `X` is the name of a variable to be bound to the `CDR` of the macro call form. Note also the use of a closing superbracket in the definition (see section 2.3).

A BBN-Lisp *substitution* macro consisted of a simple pattern (a parameter list) and a substitution template; subforms of the macro call were substituted for occurrences in the template of corresponding parameter names. A substitution macro for `ABS` would look like this:

```
((X) (COND ((GREATERP X 0) X) (T (MINUS X))))
```

However, the call `(ABS (FOO Z))` would be expanded to

```
(COND ((GREATERP (FOO Z) 0) (FOO Z)) (T (MINUS (FOO Z))))
```

leading to multiple evaluations of (FOO Z), which would be unfortunate if (FOO Z) were an expensive computation or had side effects. By way of contrast, with an open macro the call (ABS (FOO Z)) would be expanded to

```
((LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) (FOO Z))
```

which would evaluate (FOO Z) exactly once.

Despite the care sometimes required to avoid multiple evaluation, however, the pattern/template methodology for defining macros is very convenient and visually appealing. Indeed, pattern matching and template methodologies were a pervasive topic in the development of languages for Artificial Intelligence throughout the 1960's and 1970's; see section 4. We will return to the topic of template-based macros below.

Muddle [Galley, 1975], not surprisingly, had a macro facility very much like that of PDP-6 Lisp, with one slight difference. The macro expansion function, rather than being called with the macro form as its argument, was applied to the CDR of the form. This allowed Muddle's complex argument-list keywords to come into play, allowing certain simple kinds of pattern matching as for Interlisp's substitution macros:

```
<DEFMAC INC (ATM "OPTIONAL" (N 1))
  <FORM SET .ATM <FORM + <FORM LVAL .ATM> .N>>>
```

It was nevertheless necessary to laboriously construct the result as for a computed macro. The result of expanding <INC X> would be <SET X <+ .X 1>> (note that in Muddle .X is merely a readmacro abbreviation for <LVAL X>, the local value of X).

As MacLisp grew out of PDP-6 Lisp, the MacLisp community diversified, producing a variety of methodologies for defining macros. Simple macros such as INC were conceptually straightforward to write, if a bit cumbersome (certainly more clumsy than in Muddle or Interlisp):

```
(DEFUN INC MACRO (X)
  (LIST 'SETQ
        (CADR X)
        (LIST 'PLUS
              (CADR X)
              (COND ((CDDR X) (CADDR X)) (T 1)))))
```

Note that the lack of automatic decomposition ("destructuring") of the argument forms leads to many uses of CAR, CDR, and COND within the code that constructs the result. One can use LET to separate the destructuring from the construction:

```
(DEFUN INC MACRO (X)
  (LET ((VAR (CADR X))
        (N (COND ((CDDR X) (CADDR X)) (T 1))))
    (LIST 'SETQ VAR (LIST 'PLUS VAR N))))
```

but LET—itself a macro first invented and reinvented locally at each site—was a late-comer to the MacLisp world; according to Lisp Archive, it was retroactively absorbed into PDP-10 MacLisp from Lisp-Machine Lisp in 1979 at the same time as DEFMACRO and the complex Lisp Machine DEFUN argument syntax. About the best one could do during the 1970's was to use a LAMBDA expression:

```
(DEFUN INC MACRO (X)
  ((LAMBDA (VAR N)
    (LIST 'SETQ VAR (LIST 'PLUS VAR N))))
```

```
(CADR X)
(COND ((CDDR X) (CADDR X)) (T 1)))
```

and many programmers found this none too attractive. As a result, the writing of complex macros was a fairly difficult art, and wizards developed their own separate styles of macro definition. Here is some advice to the programmer from Jon L White [White, 1969–1982, May 1974]:

Suppose you have done

```
(SETQ BARODD (ARRAY NIL FIXNUM N)
      BAREVEN (ARRAY NIL FIXNUM N))
```

Now at this point, both BARODD and BAREVEN hold as value an array pointer. . . . Let us define two macros

```
(DEFUN MACRO BO (X)
  (SUBST (CADR X) 'INDEX '(ARRAYCALL FIXNUM BARODD INDEX)))

(DEFUN MACRO BE (X)
  (SUBST (CADR X) 'INDEX '(ARRAYCALL FIXNUM BAREVEN INDEX)))
```

Then we could fill BARODD and BAREVEN as follows:

```
(DO J 1 (1+ J) (> J N) (STORE (BE (1- J)) (* 2 J)))
(DO J 0 (1+ J) (NOT (< J N)) (STORE (BO J) (1+ (* 2 J))))
```

Admittedly, this saves a lot of typing. But suppose you have a host of such array variables that you would like to abbreviate with such a MACRO. Typing in all the macro definitions could be tediously repetitive. Consider the following macro-defining macro and some of its uses:

```
(DEFUN MACRO ABBA (Y)
  (SUBLIS (LIST (CONS 'SHORT (CADR Y))
                (CONS 'LONG (CADDR Y))
                (CONS 'TYPE (CADDR Y)))
    '(DEFUN MACRO SHORT (X)
      (SUBST (CDR X)
        'INDEXLIST
        '(ARRAYCALL TYPE LONG . INDEXLIST)))))
```

Now we might use ABBA to produce the macro for BE, but note that the form of the macro is slightly different—the main body of the macro output appears to be a dotted-list rather than a standard list. This is so that arrays of varying numbers of dimensions may have their abbreviations defined by the same super-macro.

```
(ABBA BO BARODD FIXNUM)
```

expands into

```
(DEFUN MACRO BO (X)
  (SUBST (CDR X)
    'INDEXLIST
    '(ARRAYCALL FIXNUM BARODD . INDEXLIST)))
```

which then causes the appropriate macro definition for BO. As you would expect, then, (BO J) expands into

```
(ARRAYCALL FIXNUM BARODD J)
```

But consider the two-dimensional hash array `HASH` defined as

```
(SETQ HASH (ARRAY NIL T 37 37))
```

Then `(ABBA HA HASH T)` defines `HA` so that `(HA 3 (+ N 2))` expands into `(ARRAYCALL T HASH 3 (+ N 2))`.

Guy Steele has accumulated a file of sophisticated macros and macro-defining macros, and the interested may consult with him about them.

To see how easily this can get out of hand, consider a macro for a simple `FOR` loop. A typical use would be this:

```
(for a 1 100
  (print a)
  (print (* a a)))
```

This should expand to

```
(do a 1 (+ a 1) (> a 100)
  (print a)
  (print (* a a)))
```

This is a trivial syntactic transformation, simple but convenient, defining a Fortran-`DO`-loop-like syntax in terms of the slightly more general “old-style” MacLisp `DO` loop (see section 3.2). In the MacLisp of the early 1970’s one might define it as follows:

```
(defun for macro (x)
  (cons 'do
    (cons (cadr x)
      (cons (caddr x)
        (cons (list '+ (cadr x) 1)
          (cons (list '> (cadr x) (caddr x))
            (cddddr x)))))))
```

That’s a lot to write for such a simple transformation.

Eventually the gurus at various MacLisp sites developed dozens of similar but not quite compatible macro-defining macros. It was not unusual for several such packages to be in use at the same site, with the adherents of each sect using whatever their wizard had developed. Such packages usually included tools for destructuring argument forms and for constructing result forms. The tools for constructing result forms fell into two major categories: substitution and pseudo-quoting. Substitution techniques required separate mention of certain symbols in a template that were to be replaced by specified values. Pseudo-quoting allowed the code to compute a replacement value to occur within the template itself; it was called pseudo-quoting because the template was surrounded by a call to an operator that was “just like `quote`” except for specially marked places within the template.

Macros took a major step forward with Lisp-Machine Lisp, which consolidated the various macro-defining techniques into two standardized features that were adopted throughout the MacLisp community and eventually into Common Lisp. The macro-defining operator `DEFMACRO` provided list-structure destructuring to arbitrary depth; the backquote feature provided a convenient and concise pseudo-quoting facility. Here is the definition of the `FOR` macro using `DEFMACRO` alone:

```
(defmacro for (var lower upper . body)
  (cons 'do
    (cons var
      (cons lower
        (cons (list '+ var 1)
          (cons (list '> var upper)
            body)))))))
```

Notice that we can name the parts of the original form. Using the backquote pseudo-quoting syntax, which makes a copy of a template, filling in each place marked by a comma with the value of the following expression, we get a very concise and easy-to-read definition:

```
(defmacro for (var lower upper . body)
  `(do ,var ,lower (+ ,var 1) (> ,var ,upper) ,@body))
```

Note the use of `,@` to indicate *splicing*.

The backquote syntax was particularly powerful when nested. This occurred primarily within macro-defining macros; because such were coded primarily by wizards, the ability to write and interpret nested backquote expressions was soon surrounded by a certain mystique. Alan Bawden of MIT acquired a particular reputation as backquote-meister in the early days of the Lisp Machine.

As an example of the nested use of backquote, consider the following version of the macro-defining macro ABBA presented above:

```
(DEFMACRO ABBA (SHORT LONG TYPE)
  `(DEFMACRO ,SHORT (&REST INDEXLIST)
    `(ARRAYCALL ,',TYPE ,',LONG ,@INDEXLIST)))
```

This is considerably shorter, and (to Bawden, at least) easier to understand. The tricky part is that idiom comma-quote-comma, which illustrates the need to have a plain quote as well as a pseudo-quote. These expressions are most easily understood from the inside out. The inner backquote expression

```
`(ARRAYCALL ,',TYPE ,',LONG ,@INDEXLIST)
```

is equivalent to

```
(LIST* 'ARRAYCALL ',TYPE ',LONG INDEXLIST)
```

which of course is the same as

```
(LIST* 'ARRAYCALL (QUOTE ,TYPE) (QUOTE ,LONG) INDEXLIST)
```

Therefore

```
`(DEFMACRO ,SHORT (&REST INDEXLIST)
  `(ARRAYCALL ,',TYPE ,',LONG ,@INDEXLIST))
```

means

```
`(DEFMACRO ,SHORT (&REST INDEXLIST)
  (LIST* 'ARRAYCALL ',TYPE ',LONG INDEXLIST))
```

which is the same as

```
(LIST 'DEFMACRO SHORT '(&REST INDEXLIST)
      (LIST 'LIST*
            ' 'ARRAYCALL
            (LIST 'QUOTE TYPE)
            (LIST 'QUOTE LONG)
            'INDEXLIST))
```

and so ABBA is the same as

```
(DEFMACRO ABBA (SHORT LONG TYPE)
  (LIST 'DEFMACRO SHORT '(&REST INDEXLIST)
        (LIST 'LIST*
              ' 'ARRAYCALL
              (LIST 'QUOTE TYPE)
              (LIST 'QUOTE LONG)
              'INDEXLIST))))
```

but this is much harder to read than the version with backquotes. For more examples of nested backquotes, see Appendix C of [Steele, 1990c].

Backquote and DEFMACRO made a big difference. This leap in expressive power, made available in a standard form, began a new surge of language extension because it was now much easier to define new language constructs in a standard, portable way so that experimental dialects could be shared. Some, including David Moon, have opined that the success of Lisp as a language designer's kit is largely due to the ability of the user to define macros that use Lisp as the processing language and list structure as the program representation, making it easy to extend the language syntax and semantics. In 1980 Kent Pitman wrote a very good summary of the advantages of macros over FEXPR's in defining new language syntax [Pitman, 1980].

Not every macro was easy to express in this new format, however. Consider the INC macro discussed above. As of November 1978, the Lisp Machine DEFMACRO destructuring was not quite rich enough to handle "optional" argument forms:

```
(DEFUN INC MACRO (VAR . REST)
  '(SETQ ,VAR (+ ,VAR ,(IF REST (CAR REST) 1))))
```

The optional part must be handled with an explicitly programmed conditional (expressed here using IF, which was itself introduced into Lisp-Machine Lisp, probably under the influence of Scheme, as a macro that expanded into an equivalent COND form). This deficiency was soon noticed and quickly remedied by allowing DEFMACRO to accept the same complex lambda-list syntax as DEFUN:

```
(DEFUN INC MACRO (VAR &OPTIONAL (N 1))
  '(SETQ ,VAR (+ ,VAR ,N)))
```

This occurred in January 1979, according to Lisp Archive, at which time MacLisp absorbed DEFMACRO and DEFUN with &-keywords from Lisp-Machine Lisp.

An additional problem was that repetitive syntax, of the kind that might be expressed in extended BNF with a Kleene star, was not captured by this framework and had to be programmed explicitly. Contemplate this simple definition of LET:

```
(defmacro let (bindings . body)
  '((lambda ,@(mapcar #'car bindings) ,@body)
    ,@(mapcar #'cadr bindings)))
```

Note the use of `MAPCAR` for iterative processing of the bindings. This difficulty was not tackled by Lisp-Machine Lisp or Common Lisp; in that community `DEFMACRO` with `&`-keywords is the state of the art today. Common Lisp did, however, generalize `DEFMACRO` to allow recursive nesting of such lambda-lists.

Further development of the theory and practice of Lisp macros was carried forward primarily by the Scheme community, which was interested in scoping issues. Macros are fraught with the same kinds of scoping problems and accidental name capture that had accompanied special variables. The problem with Lisp macros, from the time of Hart in 1963 to the mid-1980's, is that a macro call expands into an expression that is composed of symbols that have no attached semantics. When substituted back into the program, a macro expansion could conceivably take on a quite surprising meaning depending on the local environment. (Macros in other languages—the C preprocessor [Kernighan, 1978; Harbison, 1991] is one example—have the same problem if they operate by straight substitution of text or tokens.)

One practical way to avoid such problems is for the macro writer to try to choose names that the user is unlikely to stumble across, either by picking strange names such as `%%foo%%` (though it is surprising how often great minds will think alike), or by using `gensym` (as Hart did in his `select` example, shown above), or by using multiple obarrays or packages to avoid name clashes. However, none of these techniques provides an iron-glad guarantee. Steele pointed out that careful use of thunks could provably eliminate the problem, though not in all situations [Steele, 1978a].

The proponents of Scheme regarded all of these arrangements as too flawed or too clumsy for “official” adoption into Scheme. The result was that Scheme diversified in the 1980's. Nearly every implementation had some kind of macro facility but no two were alike. Nearly everyone agreed that macro facilities were invaluable in principle and in practice but looked down upon each particular instance as a sort of shameful family secret. If only The Right Thing could be found! This question became more pressing as the possibility of developing a Scheme standard was bandied about.

In the mid-1980's two new sorts of proposals were put forward: hygienic macros and syntactic closures. Both approaches involve the use of special syntactic environments to ensure that references are properly matched to definitions. A related line of work allows the programmer to control the expansion process by explicitly passing around and manipulating expander functions [Dybvig, 1986]. All of these were intended as macro facilities for Scheme, previous methods being regarded as too deeply flawed for adoption into such an otherwise elegant language.

Hygienic macros were developed in 1986 by Eugene Kohlbecker with assistance from Daniel Friedman, Matthias Felleisen, and Bruce Duba [Kohlbecker, 1986a]. The idea is to label the occurrences of variables with a tag indicating whether it appeared in the original source code or was introduced as a result of macro expansion; if multiple macro expansions occur, the tag must indicate which expansion step was involved. The technique renames variables so that a variable reference cannot refer to a binding introduced at a different step.

Kohlbecker's Ph.D. dissertation [Kohlbecker, 1986b] carried this a step further by proposing a pattern matching and template substitution language for defining macros; the underlying mechanism automatically used hygienic macro expansion to avoid name clashes. The macro-defining language was rich enough to express a wide variety of useful macros, but provided no facility for the execution of arbitrary user-specified Lisp code; this restriction was thought necessary to avoid subversion of the guarantee of good hygiene. This little language is interesting in its own right. While not as general as the separate matching and substitution facilities of `DEFMACRO` and backquote (with the opportunity to perform arbitrary computations in between), it does allow for optional and repetitive forms by using a BNF-like notation, and allows for optional situations by permitting multiple productions and using the first one that matches. For example, `INC` might be defined as

```
(extend-syntax (inc) ()
  ((inc x) (inc x 1))
  ((inc x n) (setq x (+ x n))))
```

and LET as

```
(extend-syntax (let) ()
  ((let ((var value) ...) body ...)
   ((lambda (var ...) body ...) value ...)))
```

The ellipsis “...” serves as a kind of Kleene star. Note the way in which variable-value pairs are implicitly destructured and rearranged into two separate lists in the expansion.

The first list given to **extend-syntax** is a list of keywords that are part of the macro syntax and not to be tagged as possible variable references. The second list mentions variables that may be introduced by the macro expansion but are *intended* to interact with the argument forms. For example, consider an implementation (using the Scheme call-with-current-continuation primitive) of a slight generalization of the $n + \frac{1}{2}$ loop attributed to Dahl [Knuth, 1974]; it executes statements repeatedly until its **while** clause (if any) fails or until **exit** is used.

```
(extend-syntax (loop while repeat) (exit)
  ((loop e1 e2 ... repeat)
   (call/cc (lambda (exit)
              ((label foo
                 (lambda () e1 e2 ... (foo)))))))
  ((loop e1 ... while p e2 ... repeat)
   (call/cc (lambda (exit)
              ((label foo
                 (lambda () e1 ...
                       (unless p (exit #f))
                       e2 ...
                       (foo)))))))
```

In this example **loop**, **while**, and **repeat** are keywords and should not be confused with possible variable references; **exit** is bound by the macro but is intended for use in the argument forms of the macro call. The name **foo** is not intended for such use, and the hygienic macro expander will rename it if necessary to avoid name clashes. (Note that you have to try hard to make a name available; the default is to play it safe, which makes **extend-syntax** easier and safer for novice macro writers to use.) Note the use of the idiom “**e1 e2 ...**” to require that at least one form is present if there is no **while** clause.

Syntactic closures were proposed in 1988 by Alan Bawden and Jonathan Rees [Bawden, 1988]. Their idea bears a strong resemblance to the expansion-passing technique of Dybvig, Friedman, and Haynes [Dybvig, 1986] but is more general. Syntactic contexts are represented not by the automatically managed tags of hygienic macro expansion but by environment objects; one may “close” a piece of code with respect to such a syntactic environment, thereby giving the macro writer explicit control over the correspondence between one occurrence of a symbol and another. Syntactic closures provide great power and flexibility but put the burden on the programmer to use them properly.

In 1990, William Clinger (who used to be at Indiana University) joined forces with Rees to propose a grand synthesis that combines the benefits of hygienic macros and syntactic closures, with the added advantage of running in linear rather than quadratic time. Their technique is called,

appropriately enough, “macros that work” [Clinger, 1991]. The key insight may be explained by analogy to reduction in the lambda calculus. Sometimes the rule of α -conversion must be applied to rename variables in a lambda-calculus expression so that a subsequent β -reduction will not produce a name clash. One cannot do such renaming all at once; it is necessary to intersperse renaming with the β -reductions, because a β -reduction can make two copies of a lambda-expression (hence both bind the same name) and bring the binding of one into conflict with that of the other. The same is true of macros: it is necessary to intersperse renaming with macro expansion. The contribution of Clinger and Rees was to clarify this problem and provide a fast, complete solution.

The Scheme standard [IEEE, 1991] was adopted without a macro facility, so confusion still officially reigns on this point. Macros remain an active research topic.

Why are macros so important to Lisp programmers? Not merely for the syntactic convenience they provide, but because they are programs that manipulate programs, which has always been a central theme in the Lisp community. If FORTRAN is the language that pushes numbers around, and C is the language that pushes characters and pointers around, then Lisp is the language that pushes programs around. Its data structures are useful for representing and manipulating program text. The macro is the most immediate example of a program written in a metalanguage. Because Lisp is its own metalanguage, the power of the entire programming language can be brought to bear on the task of transforming program text.

By comparison, the C preprocessor is completely anemic; the macro language consists entirely of substitution and token concatenation. There are conditionals, and one may conditionally define a macro, but a C macro may not expand into such a conditional. There is neither recursion nor metarecursion, which is to say that a C macro can neither invoke itself nor define another macro.

Lisp users find this laughable. They are very much concerned with the programming process as an object of discourse and an object of computation, and they insist on having the best possible means of expression for this purpose. Why settle for anything less than the full programming language itself?

3.4 Numerical Facilities

In Lisp 1.6 and through PDP-6 Lisp, most Lisp systems offered at most single-word fixnums (integers) and single-word flonums (floating-point numbers). (PDP-1 Lisp [Deutsch, 1985] had only fixnums; apparently the same is true of the M-460 Lisp [Hart, 1985]. Lisp 1.5 on the 7090 had floating-point [McCarthy, 1962], as did Q-32 Lisp [Saunders, 1985b] and PDP-6 Lisp [PDP-6 Lisp, 1967].)

We are still a little uncertain about origin of bignums (a data type that uses a variable amount of storage so as to represent arbitrarily large integer values, subject to the total size of the heap, which is where bignums are stored). They seem to have appeared in MacLisp and Stanford Lisp 1.6 at roughly the same time, and perhaps also in Standard Lisp. They were needed for symbolic algebra programs such as REDUCE [Hearn, 1971] and MACSYMA [Mathlab Group, 1977]. Nowadays the handling of bignums is a distinguishing feature of Lisp, though not an absolute requirement. Both the Scheme Standard [IEEE, 1991] and Common Lisp [Steele, 1990c] require them. Usually the algorithms detailed in Knuth Volume 2 are used [Knuth, 1969; Knuth, 1981]. Jon L White wrote a paper about a set of primitives that allow one to code most of bignum arithmetic efficiently in Lisp, instead of having to code the whole thing in assembly language [White, 1986].

There is also a literature on BIGFLOAT arithmetic. It has been used in symbolic algebra systems [Mathlab Group, 1977], but has not become a fixture of Lisp dialects. Lisp is often used as a platform for this kind of research because having bignums gets you 2/3 of the way there [Boehm, 1986; Vuillemin, 1988]. The MacLisp functions HAULONG and HAIPART were introduced to support

Macsyma's bigfloat arithmetic; these became the Common Lisp functions `INTEGER-LENGTH` and (by way of Lisp-Machine Lisp) `LDB`.

In the 1980's the developers of Common Lisp grappled with the introduction of the IEEE floating-point standard [IEEE, 1985]. (It is notable that, as of this writing, most other high-level programming languages have *not* grappled seriously with the IEEE floating-point standard. Indeed, ANSI X3J3 (FORTRAN) rejected an explicit request to do so.)

While Lisp is not usually thought of as a numerical programming language, there were three strong influences in that direction: MACSYMA, the S-1 project, and Gerald Sussman.

The first good numerical Lisp compiler was developed for the MACSYMA group [Golden, 1970; Steele, 1977c; Steele, 1977e]; it was important to them and their users that numerical code be both fast and compact. The result was a Lisp compiler that was competitive with the DEC PDP-10 FORTRAN compiler [Fateman, 1973].

The S-1 was initially intended to be a fast signal processor. One of the envisioned applications was detection of submarines, which seemed to require a mix of numerical signal processing and artificial intelligence techniques. The project received advice from W. Kahan in the design of its floating-point arithmetic, so it ended up being quite similar to the eventual IEEE standard. It seemed appropriate to refine the techniques of the MacLisp compiler to produce good numerical code in S-1 Lisp [Brooks, 1982b]. The S-1 offered four different floating-point formats (18, 36, 72, and 144 bits) [Correll, 1979]. Influenced by S-1 Lisp, Common Lisp provides an expanded system of floating-point data types to accommodate such architectural variation.

The inclusion of complex numbers in Common Lisp was also an inheritance from the S-1.¹ This was something of a sticking point with Scott Fahlman. A running joke was an acceptance test for nascent Common Lisp implementations developed by Guy Steele. It was in three parts. First you type `T`; if it responds `T`, it passes part 1. Second, you define the `factorial` function and then calculate

```
(/ (factorial 1000) (factorial 999))
```

If it responds 1000, it passes part 2. Third, you try `(atanh -2)`. If it returns a complex number, it passes; extra credit if it returns the *correct* complex number. It was a long time before any Common Lisp implementation passed the third part. Steele broke an implementation or two on the trade show floor with this three-part test.

Gerald Sussman and his students (including Gerald Roylance and Matthew Halfant) became interested in numerical applications and in the use of Lisp to generate and transform numerical programs [Sussman, 1988; Roylance, 1988]. Sussman also spent a fair amount of time at MIT teaching Lisp to undergraduates. Sussman thought it was absolutely crazy to have to tell students that the quotient of 10.0 and 4.0 was 2.5 but the quotient of 10 and 4 was 2. Of course, nearly all other programming languages have the same problem (Pascal [Jensen, 1974] and its derivatives being notable exceptions), but that is no excuse; Lisp aspires to better things, and centuries of mathematical precedent should outweigh the few decades of temporary aberration in the field of computers. At Sussman's urging, the `/` function was defined to return rationals when necessary, so `(/ 10 4)` in Common Lisp produces `5/2`. (This was not considered a radical change to the

¹Plans had been laid in the late 1970's to install double precision and complex numbers in PDP-10 MacLisp, but the project was never completed. Steele did a design for it and then put it aside. An undergraduate advisee of Sussman's, Barbara Kerns, eventually took on the project. Steele gave her the relevant design documents and background material, and through a stroke of misfortune she lost them the next day. Embarrassed, she plunged into her course work and spent the next four months avoiding Steele. He, buried in his own projects, never particularly noticed her absence. The next year they were married. (They spent their first anniversary in a dormitory room at Stanford, preparing overhead transparencies for their respective talks at the 1980 Lisp Conference. Ah, youth!)

language. Rational numbers were already in use in symbolic algebra systems. The developers of Common Lisp were simply integrating into the language functionality frequently required by their clients, anyway.)

All this provoked another debate, for in MacLisp and its descendants the slash was the character-quoter; moreover, backslash was the remainder operator. The committee eventually decided to swap the roles of slash and backslash, so that slash became alphabetic and backslash became the character quoter, thus allowing the division operation to be written “/” instead of “//” and allowing rational numbers to be written in conventional notation. This also solved some problems caused by a then little-known and little-loved (in the Lisp community) operating system called Unix, which used backslash as a character-quoter and slash in file names. However, it was a major incompatible change from MacLisp and Zetalisp, which left Common Lisp open to quite some criticism.)

Of course, this left Common Lisp without a truncating integer division operation, which *is* occasionally useful. Inspired by the many rounding modes of the S-1 [Correll, 1979; Hailpern, 1979] (which were influenced in turn by Kahan), Steele added *four* versions of the integer division operation to Common Lisp—`truncate`, `round`, `ceiling`, and `floor`, each of which accepts either one or two arguments and returns a quotient and remainder—thus bettering even Pascal. Overall, Common Lisp provides a much richer set of numerical primitives, and pays even closer attention to such details as the branch cuts of complex trigonometric functions, than FORTRAN ever has.

There is a problem with printing floating-point numbers. Jon L White pursued the problem for years, beginning in the early 1970’s. Guy Steele formalized the approach and their solution circulated informally in the Lisp community in the early 1980’s and was also used in at least one Pascal implementation. Steele dithered away many more years perfecting it. It was finally published in 1990 [Steele, 1990b]. (Since that time, Kent Dybvig has made further improvements to the algorithm.) By a pleasant coincidence, William Clinger had set out to solve the corresponding input problem and had submitted his solution to the same conference [Clinger, 1990], so they came out together.

On a related note, Steele put Roman numeral I/O into MacLisp in 1974 as a hack. The control variables `BASE` and `IBASE`, previously restricted to integer values between 2 and 36, could also independently be set to `ROMAN`. This feature was a bit dangerous: reading code such as

```
(do ((i 0 (+ i 1))
      (x '()) (cons (sqrt i) x)))
    ((= i 20) (return x)))
```

after `(setq ibase 'roman)` produced the equivalent of

```
(do ((i 0 (+ i 1))
      (x 10 '()) (cons (sqrt i) x)))
    ((= i 20) (return x)))
```

The initial announcement of this feature noted that not only would the value `ROMAN` result Roman numeral I/O, but you could set `BASE` or `IBASE` to `CUNEIFORM`, whereupon your Lisp would become wedged. Since that time, Roman numeral I/O has become something of a traditional hack, and became a part of the function `FORMAT` in Zetalisp and then Common Lisp.

3.5 Object-Oriented Programming

[The following text is very sketchy and needs to be filled out.]

The first blending of OOP with Lisp were message-passing-based extensions. For example, Greenblatt or Howard Cannon visited PARC in 1974 or 1975 and implemented a simple message-passing system for Lisp machine Lisp. Later Canon added multiple inheritance and the first Flavors was born [?].

Earlier, Actors [?] and Plasma [?] explored message-passing and concurrency. Scheme [Sussman, 1975b] was an attempt to understand Actors. Closures and actors were seen as the same. Actors were probably inspired by Simula (Simula I and Simula67 [?]).

Flavors came from Smalltalk [?] [?] [?] [?].

Flavors [?] [?; ?] has multiple inheritance (there was an early attempt at multiple inheritance in Smalltalk, but it was, I think, a copying system, built as an add-on library). Flavors introduced method combination in the late 1970's, which are a natural result of defining mixins. Flavors uses an automatic linearization mechanism, and the alternatives are encapsulation (explicit naming) and selection [?; ?; ?; ?] [?]. LOOPS is an Interlisp-based OOP derived from Smalltalk [?]. It had single inheritance and message-passing.

CommonLoops [Bobrow, 1986] is a derivative of LOOPS, but extending the class-ownership notion a little bit. Also it borrowed from New Flavors [Moon, 1986] [?].

Common Objects is an encapsulation-based OOP system [?]. Its encapsulation features have some similarity to those in CLU [?].

Object Lisp is a delegation-based language [Drescher, 1987], [?], [?; ?].

The Common Lisp Object System is a deliberate blending of CommonLoops and New Flavors. It introduced true generic functions and cleaned up the remnants of Smalltalk-influence on polymorphic object-oriented languages. [?; ?], [Steele, 1990c], [?].

It introduced the metaobject protocol, which treats CLOS itself as an object-oriented program that can be specialized to customize/alter the language. [?] [?; ?; ?] [?]

In Scheme there were several attempts at object-orientation. These were SCOOPS [?], OakLisp [?; ?], the delegation language done by John Ulrich for Semantic Microsystems (or Lightship as it is now called) [?]. Also see Adams and Rees [?].

In Europe we have Cointe [?], EuLisp [Padget, 1986; ?].

[End of sketchy section.]

3.6 Parallel Lisps

[The following text is very sketchy and needs to be filled out.]

These are the primary ones: Qlisp [Gabriel, 1984] [Goldman, 1988] MultiLisp [Halstead, 1984] [?], MultiScheme [?] [?]. Improvements to Qlisp propositional parameters [?].

Also more theoretical stuff like [?], [?] [?]

Linda-influenced stuff [?]. There is SPUR [?], Curare [?]. There is automatic parallelization [?; ?].

[End of sketchy section.]

3.7 Some Notable Failures

Despite Lisp's tendency to absorb new features over time, both from other programming languages and from experiments within the Lisp community, there are a few ideas that have been tried repeatedly in various forms but for some reason simply don't catch on in the Lisp community. Notable among these ideas are Algol-style syntax, generalized multiple values, and logic programming with unification of variables.

3.7.1 Algol-Style Syntax

Ever since Steve Russell first hand-coded an implementation of **EVAL**, S-expressions have been the standard notation for writing programs. In almost any Lisp system of the last thirty years, one could write the function **UNION** (which computes the union of two sets represented as lists of elements) in roughly the following form:

```
(defun union (x y)
  (cond ((null x) y)
        ((member (car x) y) (union (cdr x) y))
        (t (cons (car x) (union (cdr x) y)))))
```

The original intention, however, in the design of Lisp was that programs would be written as M-expressions; the S-expression syntax was intended only for representation of data. The **UNION** function in M-expression notation looks like this:

```
union[x;y] = [null[x]→y;
              member[car[x];y]→union[cdr[x];y];
              T→cons[car[x];union[cdr[x];y]]]
```

But as McCarthy noted [McCarthy, 1981]:

The unexpected appearance of an interpreter tended to freeze the form of the language.... The project of defining M-expressions precisely... was neither finalized nor completely abandoned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred internal notation [i.e., S-expressions] to any Fortran-like or Algol-like notation that could be devised.

Yet that was not the end of the story. Since that time there have been many other efforts to provide Lisp with an Algol-like syntax. Time and again a Lisp user or implementor has felt a lack in the language and provided a solution—and not infrequently attracted a substantial group of users—and yet in the long run none of these has achieved acceptance.

The earliest example of this—after M-expressions, of course—appears to have been Henneman's A-language [Henneman, 1985]. Henneman gives the following definition of **UNION**:

```
(DEFINE UNION (OF AND) (8)
  (UNION OF X AND Y) (IF X IS
    EMPTY THEN Y ELSE IF FIRST OF
    X IS A MEMBER OF Y THEN UNION
    OF REST OF X AND Y ELSE
    CONNECT FIRST OF X TO BEGIN
    UNION OF REST OF X AND Y
  END))
```

The number 8 in this definition is the precedence of the **UNION** operator; note the use of **BEGIN** and **END** as parenthetical delimiters, necessary because the **CONNECT ... TO ...** operator (which means **CONS**) has higher precedence.

We find it curious that Henneman went to the trouble of pretty-printing the M-expressions and S-expressions in his paper but presented all his examples of A-language in the sort of run-on, block-paragraph style often seen in the S-expressions of his contemporaries. Nowadays we would format such a program in this manner for clarity:

```
(DEFINE UNION (OF AND) (8)
  (UNION OF X AND Y)
  (IF X IS EMPTY THEN Y
    ELSE IF FIRST OF X IS A MEMBER OF Y
      THEN UNION OF REST OF X AND Y
    ELSE CONNECT FIRST OF X TO
      BEGIN UNION OF REST OF X AND Y END
  ))
```

Such formatting was not unheard of; the Algol programmers of the day used similar indentation conventions in their published programs.

[The following text is very sketchy and needs to be filled out.]

Lisp 2.0

[End of sketchy section.]

The EL1 language was designed by Ben Wegbreit as part of his Ph.D. research [Wegbreit, 1970]. It may be loosely characterized as a Lisp with an Algol-like surface syntax and strong data typing. A complete programming system called ECL was built around EL1 at Harvard in the early 1970's [Wegbreit, 1971; Wegbreit, 1972; Wegbreit, 1974]. The UNION function in EL1 looks like this:

```
union <- EXPR(x: FORM, y: FORM; FORM)
  [] x=NIL => y;
  MEMBER(CAR(x), y) => union(CDR(x), y);
  CONS(CAR(X), union(CDR(x), y)) [];
```

Note the type declarations of *x*, *y*, and the result as type FORM (pointer to dotted pair). The digraphs [] and [] are equivalent to BEGIN and END (they looked better on a Model 33 Teletype than they do here). Within a block the arrow => indicates the conditional return of a value from the block, resulting in a notation reminiscent of McCarthy's conditional notation for M-expressions.

Lisp itself was not widely used at Harvard's Center for Research in Computing Technology at that time; EL1 and PPL (Polymorphic Programming Language, a somewhat more Joss-like interactive system) may have been Harvard's answer to Lisp at the time. ECL might have survived longer if Wegbreit had not left Harvard for Xerox in the middle of the project. As it was, ECL was used for research and course work at Harvard throughout the 1970's.

We have already discussed Teitelman's CLISP (Conversational Lisp), which was part of Interlisp [Teitelman, 1974]. The function UNION was built into Interlisp, but could have been defined using CLISP in this manner:

```
DEFINEQ((UNION (LAMBDA (X Y)
  (IF ~X THEN Y
    ELSEIF X:1 MEMBER Y THEN UNION X::1 Y
    ELSE <X:1 !(UNION X::1 Y)>])
```

In CLISP, ~ is a unary operator meaning NOT or NULL. *X:n* is element *n* of the list *X*, so *X:1* means (CAR *X*); similarly *X::1* means (CDR *X*). The function MEMBER is predefined by CLISP to be an infix operator, but UNION is not (though the user may so define it if desired). Angle brackets indicate construction of a list; ! within such a list indicates splicing, so <A !B> means (CONS A B). Finally, the use of a final] to indicate the necessary number of closing parentheses (four, in this case), while not a feature of CLISP proper, is consistent with the Interlisp style.

MLISP was an Algol-like syntax for Lisp, first implemented for the IBM 360 by Horace Enea and then re-implemented for the PDP-10 under Stanford Lisp 1.6 [Smith, 1970]. It provided infix

operators; a complex **FOR** construct for iteration; various subscripting notations such as **A(1,3)** (element of a two-dimensional array) and **L[1,3,2]** (equivalent to **(cadr (caddr (car L)))**); “vector” operations (a concise notation for **MAPCAR**); and destructuring assignment.

```
EXPR UNION (X,Y);                                %MLISP version of UNION
  IF  $\neg$ X THEN Y ELSE
  IF  $X[1] \in Y$  THEN UNION( $X\downarrow 1$ ,Y)
  ELSE  $X[1]$  CONS UNION( $X\downarrow 1$ ,Y);
```

Vaughan Pratt developed an Algol-style notation for Lisp called CGOL [Pratt, 1973]. Rather than embedding algebraic syntax within S-expressions, CGOL employed a separate full-blown tokenizer and parser. This was first implemented for Stanford Lisp 1.6 in 1970 when Pratt was at Stanford; at this time there was an exchange of ideas with the MLISP project. After Pratt went to MIT shortly thereafter, he implemented a version for MacLisp [Pratt, 1976]. Versions of this parser were also used in the symbolic algebra systems SCRATCHPAD at IBM Yorktown and MACSYMA at MIT’s Project MAC; Fred Blair, who also developed LISP370, did the reimplementations for SCRATCHPAD, while Michael Genesereth did it for MACSYMA.

Our CGOL version of the **UNION** function defines it as an infix operator (the numbers 14 and 13 are left and right “binding powers” for the parser):

```
define x "UNION" y, 14, 13;
  if not x then y
  else if member(car x, y) then cdr x union y
  else car x . cdr x union y  $\diamond$ 
```

Here we have assumed the version of CGOL implemented at MIT, which stuck to the standard ASCII character set; the exact same definition using the Stanford extended character set would be:

```
define x "U" y, 14, 13;
  if  $\neg$ x then y else if  $\alpha x \in y$  then  $\beta x \cup y$  else  $\alpha x . \beta x \cup y \diamond$ 
```

The “.” represents **CONS** in both the above examples; the delimiter $\diamond$ (actually the ASCII “alt-mode” character, nowadays called “escape”) indicates the end of a top level expression. All unary Lisp functions are unary operators in CGOL, including **CAR** and **CDR**. In the definition above we have relied on the fact that such unary operators have very high precedence, so **cdr x union y** means **(cdr x) union y**, not **cdr (x union y)**. We also carefully chose the binding powers for **UNION** relative to “.” so that the last expression would be parsed as **(car x) . ((cdr x) union y)**. It is not obvious that this is the best choice; Henneman chose to give **CONS** (in the form of **CONNECT ... TO ...**) higher precedence than **UNION**. Pratt remarked [Pratt, 1976]:

If you want to use the CGOL notation but don’t want to have anything to do with binding powers, simply parenthesize every CGOL expression as though you were writing in Lisp. However, if you omit all parentheses ... you will not often go wrong.

Compare this to Henneman’s remark [Henneman, 1985]:

The one great cause of most of the incorrect results obtained in practice is an incorrect precedence being assigned to a function.

During the 1970’s a number of “AI languages” were designed to provide specific programming constructs then thought to be helpful in writing programs for AI applications. Some of these were embedded within Lisp and therefore simply inherited Lisp syntax (and in some cases influenced Lisp

syntax—see section 4 for a discussion of these). Those that were not embedded usually had a syntax related to that of Algol, while including some of the other features of Lisp (such as symbolic data structures and recursive functions). Among these were POP-2 [Burstall, 1971], SAIL [Feldman, 1972], and the Pascal-based TELOS [Travis, 1977].

The idea of introducing Algol-like syntax into Lisp keeps popping up and has seldom failed to create enormous controversy between those who find the universal use of S-expressions a technical advantage (and don't mind the admitted relative clumsiness of S-expressions for numerical expressions) and those who are certain that algebraic syntax is more concise, more convenient, or even more *natural* (whatever that may mean, considering that all these notations are artificial).

We conjecture that Algol-style syntax has not really caught on in the Lisp community as a whole for two reasons. First, there are not enough special symbols to go around. When your domain of discourse is limited to numbers or characters, there are only so many operations of interest, and it is not difficult to assign one special character to each and be done with it. But Lisp has a much richer domain of discourse, and a Lisp programmer often approaches an application as yet another exercise in language design; the style typically involves designing new data structures and new functions to operate on them—perhaps dozens or hundreds—and it's just too hard to invent that many distinct symbols (though the APL community certainly has tried). Ultimately one must always fall back on a general function-call notation; it's just that Lisp programmers don't wait until they fail.

Second, and perhaps more important, Algol-style syntax makes programs look less like the data structures used to represent them. In a culture where the ability to manipulate representations of programs is a central paradigm, a notation that distances the appearance of a program from the appearance of its representation as data is not likely to be warmly received (and this was, and is, one of the principal objections to the inclusion of `loop` in Common Lisp).

On the other hand, precisely because Lisp makes it easy to play with program representations, it is always easy for the novice to experiment with alternative notations. Therefore we expect future generations of Lisp programmers to continue to reinvent Algol-style syntax for Lisp, over and over and over again, and we are equally confident that they will continue, after an initial period of infatuation, to reject it. (Perhaps this process should be regarded as a rite of passage for Lisp hackers.)

3.7.2 Generalized Multiple Values

Many Lisp extenders have independently gone down the following path. Sometimes it is desirable to return more than one item from a function. It is awkward to return some of the results through global variables, and inefficient to cons up a list of the results (pushing the system that much closer to its next garbage collection) when we know perfectly well that they could be returned in machine registers or pushed onto the machine control stack. Curiously, the prototypical example of a function that ought to return two results is not symbolic but numerical: integer division might conveniently return both a quotient and a remainder. (Actually, it would be just as convenient for the programmer to use two separate functions, but we know perfectly well that the computation of one produces the other practically for free—again it is an efficiency issue.)

Suppose, then, that some primitive means of producing multiple values is provided. One way is to introduce new functions and/or special forms. Common Lisp, for example, following Lisp-Machine Lisp, has a primitive function called `VALUES`; the result of `(values 3 4 5)` is the three numbers 3, 4, and 5. The special form

```
(multiple-value-bind (p q r) (foo) body)
```

executes its *body* with the variables *p*, *q*, and *r* locally bound to three values returned as the value of the form (*foo*).

But this is all so ad hoc and inelegant. Perhaps multiple values can be made to emerge from the intrinsic structure of the language itself. Suppose, for example, that the body of a lambda expression were an implicit **VALUES** construct, returning the values of all its subforms, rather than an implicit **PROGN**, returning the values of only the last subform? That takes care of producing multiple values. Suppose further that function calls were redefined to use all the values returned by each subform, rather than just one value from each subform? Then one could write

```
((lambda (quo rem) ...) (/ 44 6))
```

thereby binding *quo* to the quotient 7 and *rem* to the remainder 2. That takes care of consuming multiple values. All very simple and tidy! Oops, two details to take care of. First, returning to lambda expressions, we see that, for consistency, they need to return all the values of all the subforms, not just one value from each subform. So the form

```
((lambda (quo rem) (/ quo rem) (/ rem quo)) (/ 44 6))
```

returns four values: 3, 1, 0, and 2. Second, there is still a need for sequencing forms that have side effects such as assignment. The simple solution is to make such forms as (*setq* *x* 0) and (*print* *x*) return zero values, so that

```
((lambda (quo rem) (print quo) rem) (/ 44 6))
```

returns only the remainder 2 after printing the quotient 7. (This stone kills another bird at the same time by eliminating the annoyance of an interactive top-level printing hundreds of lines of NIL after NIL just because of

```
(setq my-array (make-array 1000000))
```

or some such interactive initialization step.)

This all has a very simple and attractive stack-based implementation. Primitives simply push all their values, one after another, onto the stack. At the start of the processing for a function call, place a marker on the stack; after all subforms have been processed, simply search the stack for the most recent marker; everything above it should be used as arguments for the function call—but be sure to remove or cancel the marker before transferring control to the function. (An alternative strategy uses a shallow-bound counter instead of markers; this avoids the search, at the expense of updating the counter as values are pushed.) It is gratifying that no special action is required for processing the bodies of lambda-expressions.

Yes, this is all very neat and tidy—and as soon as you try to use it, you find that code becomes much, much harder to understand, both for the maintainer and for the compiler. Even if the programmer has the discipline not to write

```
(cons (/ 44 6) (setq x 0))
```

which returns (7 . 2) after setting *x* to 0, the compiler can never be sure that no such atrocities lurk in the code it is processing. In the absence of fairly complete information about how many values are produced by each function, including all user-defined functions, a compiler cannot verify that a function call will supply the correct number of arguments. An important practical check for Lisp programming errors is thus made all but impossible.

Conditionals introduce two further problems. First: what shall be the interpretation of

```
(if (foo) (bar))
```

if (foo) returns two values? Shall the second value be discarded, or treated as the value to be returned if the first value is true? Perhaps the predicate should be required to return exactly one value. Very well, but there remains the fact that the two subforms might return different numbers of values; (if (foo) 3 (/ 44 6)) might return the single value 3 or the multiple values 7 and 2. It follows immediately that no compiler, even if presented with a complete program, can deduce in general how many values are returned by each function call; it is formally undecidable.

Now all this might seem to be entirely in the spirit of Lisp as a weakly typed language; if the types of the values returned by functions may not be determinable until run time, why not their very cardinality? And declarations might indicate the number of values where efficiency is important, just as type declarations already assist many Lisp compilers. Nevertheless, as a matter of fact, nearly everyone who has followed this path has given up at this point in the development, muttering “This way madness lies,” and returned home rather than fall into the tarpit.

We ourselves have independently followed this line of thought and have had conversations with quite a few other people who have also done so. There are few published sources we can cite, however, precisely because most of them eventually judged it a bad idea before publishing anything about it. (This is not to say that it actually *is* a bad idea, or that some variation cannot eliminate its disadvantages; here we wish merely to emphasize the similarity of thinking among many independent researchers.) Among the most notable efforts that did produce actual implementations before eventual abandonment are SEUS and POP-2 [Burstall, 1971]. The designers and implementors of SEUS (Richard Weyhrauch, Carolyn Talcott, David Posner, Ralph Goren, William Scherlis, Len Bosack, and Gabriel) never published their results, although it was a novel language design, and had a fast, compiler- and microcode-based implementation, complete with programming environment. POP-2 was regarded by its designers as an AI language, one of the many produced in the late 1960’s and early 1970’s, rather than as a variant of Lisp; it enjoyed quite some popularity in Europe and was used to implement the logic programming language POPLOG [Mellish, 1984].

3.7.3 Logic Programming and Unification

During the 1970’s and on into the 1980’s there have been a number of attempts to integrate the advantages of the two perhaps foremost AI programming language families, Lisp and Prolog, into a single language. Such efforts were particularly a feature of the software side of the Japanese Fifth Generation project. Examples of this are Robinson’s LOGLISP [Robinson, 1982], the TAO project [Takeuchi, 1983; Okuno, 1984], and TABLOG [Malachi, 1984]. There have also been related attempts to integrate functional programming and Prolog. (All these should be contrasted with the use of Lisp as a convenient language for *implementing* Prolog, as exemplified by Komorowski’s QLOG [Komorowski, 1982] and the work of Kahn and Carlsson [Kahn, 1984].)

We conjecture that this idea has not caught on in the Lisp community because of unification, the variable-matching process used in Prolog. Indeed one can easily design a language that has many of the features of Lisp but uses unification during procedure calls. The problem is that unification is sufficiently different in nature from lambda-binding that the resulting language doesn’t really feel like Lisp any more. To the average Lisp programmer, it feels like an extension of Prolog but not an extension of Lisp; you just can’t mess around that much with something as fundamental as procedure calls. On the other hand, one can leave Lisp procedure calls as they are and provide unification as a separate facility that can be explicitly invoked. But then it is just another Lisp library routine, and the result doesn’t feel at all like Prolog.

3.8 Other Topics

Other topics we might usefully address are:

- continuations as an expressive device
- reification and reflection
- garbage collection (high points include mark/sweep, copying, flirtations with reference counting, Baker's real-time methods [Baker, 1978], Bishop's proposal to handle a very large address space, White's suggestion "don't do it!" [White, 1980], Moon's ephemeral GC [Moon, 1984]; Brooks on GC for stock hardware; also mention Cohen's paper in *Computing Surveys*)
- various techniques for managing stacks efficiently: spaghetti stacks [Bobrow, 1973], [Steele, 1977a], McDermott's ideas [McDermott, 1980], Steele and Sussman on lazy register saving [Steele, 1980], stack allocation of numbers [Steele, 1977e] [Brooks, 1982b], Stallman's phantom stacks, Appel's claim (*Info. Proc. Letters* 25, 275–279) that GC is faster than stack allocation, also [Clinger, 1988], [Bartley, 1986] [Hieb, 1990]
- record structures, DEFSTRUCT, Interlisp RECORD package
- development of oblist to obarray to packages to modules ...
- Hash tables: speed, creating relationships between objects while also regarding them as opaque (don't have to modify either of them to create a pointer), decreased use of property lists, weak hash links that don't prevent an object from being reclaimed (hence a hash table with weak links serves as a cache)
- Talk about programming environment strategies: File-based versus memory-based; Pretty-printing; Lisp-based editor, esp. structure editing, vs. EMACS; Influence of mechanical pretty-printing on programming style; Buying into a set of tools; Languages features and hooks that make tools possible: Trace, advice, inspect, evalhook, etc.
- Closures (dynamic and lexical—trace back to Moses 1970)
- Nonlocal exits: `errset/err`, `catch`, `call/cc`, `unwind-protect`, whatever Interlisp had, `GO` to an outer `PROG`, lexically scoped `GO` an `RETURN` (Common Lisp), continuations as a general language feature and implications for implementation

4 Lisp as a Language Laboratory

An interesting aspect of the Lisp culture, in contrast to those surrounding most other programming languages, is that toy dialects are regarded with a fair amount of respect. Lisp has been used throughout its history as a language laboratory. It is trivial to add a few new functions to Lisp in such a way that they look like system-provided facilities. Given macros, CLISP, or the equivalent, it is pretty easy to add new control structures or other syntactic constructs. If that fails, it is the work of only half an hour to write, in Lisp, a complete interpreter for a new dialect. To see how amazing this is, imagine starting with a working C, Fortran, Pascal, PL/I, BASIC, or APL system—you can write and run any program in that language, but have no access to source code for the compiler or interpreter—and then tackling these three exercises:

1. Add a new arithmetic operator to the language similar to the one for Pythagorean addition in Knuth's Metafont language [Knuth, 1986]: `a++b` computes $\sqrt{a^2 + b^2}$. The language is to be augmented in such a way that the new operator is syntactically similar to the language operators for addition and subtraction. (For C, you may use `+++` or `@` rather than `++`; for APL, use one of the customary awful overstrikes such as $\boxplus$.)

2. Add a **case** statement to the language. (If it already has a **case** statement, then add a statement called **switch** that is just like the **case** statement already in the language, except that when the selected branch has been executed control falls through to succeeding branches; a special **break** statement, or dropping out of the last branch, must be used to terminate execution of the **switch** statement.)
3. Add full lexically scoped functional closures to the language.

Without source code for the compiler or interpreter, all three projects require one practically to start over from scratch. That is part of our point. But even given source code for a Fortran compiler or APL interpreter, all three exercises are much more difficult than in Lisp. The Lisp answer to the first one is a one-liner (shown here in Common Lisp):

```
(defun ++ (x y) (+ (sqrt x) (sqrt y)))
```

Lisp does not reserve special syntax (such as infix operators) for use by built-in operations, so user-defined functions look just like system-defined functions to the caller.

The second requires about a dozen lines of code (again, in Common Lisp, using backquote syntax as described in the discussion of macros):

```
(defmacro switch (value &rest body)
  (let* ((newbody (mapcar #'(lambda (clause)
                              '(',(gensym) ,@(rest clause)))
                          body))
        (switcher (mapcar #'(lambda (clause newclause)
                              '(',(first clause) (go ,(first newclause)))))
        body newbody)))
  '(block switch
    (tagbody (case ,value ,@switcher)
              (break)
              ,@(apply #'nconc newbody))))

(defmacro break () '(return-from switch))
```

Here we use two macros, one for **switch** and one for **break**, which together cause the statement

```
(switch n
  (0 (princ "none") (break))
  (1 (princ "one "))
  (2 (princ "too "))
  (3 (princ "many")))
```

(which always prints either *many*, *too many*, *one too many*, *none*, or *nothing*) to expand into

```
(block switch
  (tagbody (case n
              (0 (go G0042))
              (1 (go G0043))
              (2 (go G0044))
              (3 (go G0045)))
    (return-from switch)
    G0042 (princ "none")
```

```

                (return-from switch)
G0043  (princ "one ")
G0044  (princ "too ")
G0045  (princ "many"))))

```

which is not unlike the code that would be produced by a C compiler.

For examples of interpreters that solve the third problem in about 100 lines of code, see [Steele, 1978c; Steele, 1978b].

There is a rich tradition of experimenting with augmentations of Lisp, ranging from “let’s add just one new feature” to inventing completely new languages using Lisp as an implementation language. This activity was carried on particularly intensively at MIT during the late 1960’s and early 1970’s but also at other institutions such as Stanford University, Carnegie-Mellon University, and Indiana University. At that time it was customary to make a set of ideas about programming style concrete by putting forth a new programming language as an exemplar. (This was true outside the Lisp community as well; witness the proliferation of Algol-like, and particularly Pascal-inspired, languages around the same time period. But Lisp made it convenient to try out little ideas with a small amount of overhead, as well as tackling grand revampings requiring many man-months of effort.)

One of the earliest Lisp-based languages was METEOR [Bobrow, 1985], a version of COMIT (note the pun) with Lisp syntax. COMIT [MIT RLE, 1962a; MIT RLE, 1962b; Yngve, 1972] was a pattern-matching language that repeatedly matched a set of rules against the contents of a flat, linear workspace of symbolic tokens; it was a precursor of SNOBOL and an ancestor of such rule-based languages as OPS5 [Forgy, 1977]. METEOR was embedded within the MIT Lisp 1.5 system that ran on the IBM 7090. The Lisp code for METEOR is a little under 300 80-column cards (some with more whitespace than others). By contrast, the 7090 implementation of the COMIT interpreter occupied about 10,000 words or memory, according to Yngve; assuming this reflects about 10,000 lines of assembly language code, we can view this as an early example of the effectiveness of LISP as a high-level language for prototyping other languages.

Another early LISP-based pattern-matching language was CONVERT [Guzman, 1966]. Whereas METEOR was pretty much a straight implementation of COMIT represented as Lisp data structures, CONVERT merged the pattern-matching features of COMIT with the recursive data structures of Lisp, allowing the matching of recursively defined patterns to arbitrary Lisp data structures.

Carl Hewitt designed an extremely ambitious Lisp-like language for theorem-proving called Planner [Hewitt, 1972]. Its primary contributions consisted of advances in pattern-directed invocation and the use of automatic backtracking as an implementation mechanism for goal-directed search. It was never completely implemented as originally envisioned, but it spurred three other important developments in the history of Lisp: Micro-Planner, Muddle, and Conniver.

Gerald Jay Sussman, Drew McDermott, and Eugene Charniak implemented a subset of Planner called Micro-Planner [Sussman, 1971], which was embedded within the MIT PDP-6 Lisp system that eventually became MacLisp. The semantics of the language as implemented were not completely formalized. The implementation techniques were rather ad hoc and did not work correctly in certain complicated cases; the matcher was designed to match two patterns, each of which might contain variables, but did not use a complete unification algorithm. (Much later, Sussman, on learning about Prolog, remarked to Guy Steele that Prolog appeared to be the first correct implementation of Micro-Planner.)

A version of Planner was also implemented in POP-2 [Davies, 1984].

The language Muddle (later MDL) was an extended version of Lisp and in some ways a competitor, designed and used by the Dynamic Modeling Group at MIT, which was separate from the MIT AI Laboratory but in the same building at 545 Technology Square. This effort was begun in late 1970 by Gerald Jay Sussman, Carl Hewitt, Chris Reeve, and David Cressey, later joined by Bruce Daniels, Greg Pfister, and Stu Galley. It was designed "... as a successor to Lisp, a candidate vehicle for the Dynamic Modeling System, and a possible base for implementation of Planner-70." [Galley, 1975] To some extent the competition between Muddle and Lisp, and the fact that Sussman had a foot in each camp, resulted in cross-fertilization. The I/O, interrupt handling, and multiprogramming (that is, coroutining) facilities of Muddle were much more advanced than those of MacLisp at the time. Muddle had a more complex garbage collector than PDP-10 MacLisp ever had, as well as a larger library of application subroutines, especially for graphics. (Some Lisp partisans at the time would reply that Muddle was used entirely to code libraries of subroutines but no main programs! But in fact some substantial applications were coded in Muddle.) Muddle introduced the lambda-list syntax markers OPTIONAL, REST, and AUX that were later adopted by Conniver, Lisp Machine Lisp, and Common Lisp.

The language Conniver was designed by Drew McDermott and Gerald Jay Sussman in 1972 in reaction to perceived limitations of Micro-Planner and in particular of its control structure. In the classic paper *Why Conniving Is Better Than Planning* [Sussman, 1972b; Sussman, 1972a], they argued that automatic nested backtracking was merely an overly complicated way to express a set of FORALL loops used to perform exhaustive search:

It is our contention that the backtrack control structure that is the backbone of Planner is more of a hindrance in the solution of problems than a help. In particular, automatic backtracking encourages inefficient algorithms, conceals what is happening from the user, and misleads him with primitives having powerful names whose power is only superficial.

The design of Conniver put the flow of control very explicitly in the hands of the programmer. The model was an extreme generalization of coroutines; there was only one active locus of control, but arbitrarily many logical threads and primitives for explicitly transferring the active locus from one to another. This design was strongly influenced by the "spaghetti stack" model introduced by Daniel Bobrow and Ben Wegbreit [Bobrow, 1973] and implemented in BBN-Lisp (later to be known as Interlisp). Like spaghetti stacks, Conniver provided separate notions of a data environment and a control environment and the possibility of creating closures over either. (Later work with the Scheme language brought out the point that data environments and control environments do not play symmetrical roles in the interpretation of Lisp-like languages [Steele, 1977a].) Conniver differed from spaghetti stacks in ways stemming primarily from implementation considerations. The main point of Conniver was generality and ease of implementation; it was written in Lisp and represented control and data environments as Lisp list structures, allowing the Lisp garbage collector to handle reclamation of abandoned environments. The implementation of spaghetti stacks, on the other hand, involved structural changes to a Lisp system at the lowest level. It addressed efficiency issues by allowing stack-like allocation and deallocation behavior wherever possible. The policy was pay as you go but don't pay if you don't use it: programs that do not create closures should not pay for the overhead of heap management of control and data environments.

[The following text is very sketchy and needs to be filled out.]

Need to talk here about pattern-matching and backtracking languages outside MIT, such as MLISP2 [Smith, 1973], LISP70 [Tesler, 1973], QA4, the LEAP extensions to the SAIL language [Swinehart, 1972; Feldman, 1972], and PLITS. Need to check 1977 AIPL Conference in Rochester.

[End of sketchy section.]

At about this time Carl Hewitt and his students began to develop the actor model of computation, in which every computational entity, whether program or data, is an actor: an agent that can receive and react to messages. The under-the-table activity brought out by Conniver was made even more explicit in this model; everything was message-passing, everything ran on continuations. Hewitt and his student Brian Smith commented on the interaction of a number of research groups at the time [?]:

The early work on PLANNER was done at MIT and published in IJCAI-69 [Hewitt, 1969]. In 1970 a group of interested researchers (including Peter Deutsch, Richard Fikes, Carl Hewitt, Jeff Rulifson, Alan Kay, Jim Moore, Nils Nilsson, and Richard Waldinger) gathered at Pajaro Dunes to compare notes and concepts. . . .

In November 1972, Alan Kay gave a seminar at MIT in which he emphasized the importance of using intentional definitions of data structures and of passing messages to them such as was done to a limited extent for the “procedural data structures” in the lambda calculus languages of Landin, Evans, and Reynolds and extensively in SIMULA-67. His argument was that only the data type itself really “knows” how to implement any given operation. We had previously given some attention to procedural data structures in our own research. . . . However, we were under the misconception that procedural data structures were too inefficient for practical use although they had certain advantages.

Kay’s lecture struck a responsive note . . . We immediately saw how to use his idea . . . to extend the principle of procedural embedding of knowledge to data structures. In effect each type of data structure becomes a little *plan* of what to do for each kind of request that it receives. . . .

Kay proposed a language called SMALLTALK with a token stream oriented interpreter to implement these ideas. . . .

[At that time,] Peter Bishop and Carl Hewitt were working to try to obtain a general solution to the control structure problems which had continued to plague PLANNER-like problem solving systems for some years. Sussman had proposed a solution oriented around “possibility lists” which we felt had very serious weaknesses. . . . Simply looking at their contents using `try-next` can cause unfortunate global side-effects . . . [which] make Conniver programs hard to debug and understand. The token streams of SMALLTALK have the same side-effect problem as the possibility lists of Conniver. After the lecture, Hewitt pointed out to Kay the control structure problems involved in his scheme for a token stream oriented interpreter.

By December 1972, we succeeded in generalizing the message mechanism of SMALLTALK and SIMULA-67; the port mechanism of Krutar, Balzer, and Mitchell; and the previous CALL statement of PLANNER-71 to a universal communication mechanism. Our generalization solved the control structure problems that Hewitt pointed out to Kay in the design of SMALLTALK. We developed the actor transmission communication primitive as part of a new language-independent, machine-independent, behavioral model of computation. The development of the actor model of computation and its ramifications is our principal original contribution to this area of research. . . .

The following were the main influences on the development of the actor model of computation:

- The suggestion by [Alan] Kay that procedural embedding be extended to cover data structures in the context of our previous attempts to generalize the work by Church, Landin, Evans, and Reynolds on “functional data structures.”
- The context of our previous attempts to clean up and generalize the work on corou-

tine control structures of Landin, Mitchell, Krutar, Balzer, Reynolds, Bobrow-Wegbreit, and Sussman.

- The influence of Seymour Papert’s “little man” metaphor for computation in LOGO.
- The limitations and complexities of capability-based protection schemes. Every actor transmission is in effect an inter-domain call efficiently providing an intrinsic protection on actor machines.
- The experience developing previous generations of PLANNER. Essentially the whole PLANNER-71 language (together with some extensions) was implemented by Julian Davies in POP-2 at the University of Edinburgh.

In terms of the actor model of computation, control structure is simply a pattern of passing messages. . . . Actor control structure has the following advantages over that of Conniver:

- A serious problem with the Conniver approach to control structure is that the programmer (whether human or machine) must think in terms of low level data structures such as activation records or possibility links. The actor approach allows the programmer to think in terms of the behavior of objects that naturally occur in the domain being programmed. . . .
- Actor transmission is entirely free of side-effects. . . .
- The control mechanisms of Conniver violate principles of modularity. . . . Dijkstra has remarked that the use of the `goto` is associated with badly structured programs. We concur in this judgement but feel that the reason is that the `goto` is not a sufficiently powerful primitive. The problem with the `goto` is that a message cannot be sent along with control to the target. . . .
- Because of its primitive control structures, Conniver programs are difficult to write and debug. . . . Conniver programs are prone to going into infinite loops for no reason that is very apparent to the programmer.

Nevertheless Conniver represents a substantial advance over Micro-Planner in increasing the generality of goal-oriented computations that can be easily performed. However, this increase in generality comes at the price of lowering the level of the language of problem solving. It forces users to think in low level implementation terms such as “possibility lists” and “a-links.” We propose a shift in the paradigm of problem solving to be one of a society of individuals communicating by passing messages.

We have quoted Smith and Hewitt at length for three reasons: because their comparative analysis is very explicit; because the passage illustrates the many connections among different ideas floating around in the AI, Lisp, and other programming language communities; and because this particular point in the evolution of ideas represented a distillation that soon fed back quickly and powerfully into the evolution of Lisp itself.

Hewitt and his students (notably Howie Shrobe, Brian Smith, Todd Matson, Roger Hale, Peter Bishop, Marilyn McLennan, Russ Atkinson, Mike Freiling, Ken Kahn, Keith Nishihara, Kathy Van Sant, Aki Yonizawa, Benjamin Kuipers, Richard Stieger, and Irene Greif) developed and implemented in MacLisp a new language to make concrete the actor model of computation. This language was first called Planner-73 but the name was later changed to PLASMA (PLAnner-like System Modeled on Actors) [?; Hewitt, 1975].

While the syntax of PLASMA was recognizably Lisp-like, it made use of several kinds of parentheses and brackets (as did Muddle) as well as many other special characters. It is reasonable to assume that Hewitt was tempted by the possibilities of the then newly available Knight keyboard and Xerox Graphics Printer (XGP) printer. (The Xerox Graphics Printer was the first of the laser printers, 200 dots per inch. The keyboards, designed by Tom Knight of the MIT AI Lab, were the MIT equivalent of the extended-ASCII keyboards developed years earlier at the Stanford AI Laboratory. Like the Stanford keyboards, Knight keyboards had Control and Meta keys (which were soon pressed into service in the development of the command set for the EMACS text editor) and a set of graphics that included such exotic characters as α , β , and $\equiv$. The XGP, at 200 dots per inch, made possible the printing of such exotic characters.) The recursive factorial function looked like this in PLASMA:

```
[factorial  $\equiv$ 
  (cases
    ( $\equiv$  [0] 1)
    ( $\equiv$  [ $=n$ ] (n * (factorial (n - 1))))))]
```

Note the use of infix arithmetic operators. This was not merely clever syntax, but clever semantics; ($n - 1$) really meant that a message containing the subtraction operator and the number 1 was to be sent to the number/actor/object named by n .

One may argue that Lisp development at MIT took two distinct paths during the 1970's. In the first path, MacLisp was the workhorse tool, coded in assembly language for maximum efficiency and compactness, serving the needs of the AI Laboratory and the MACSYMA group. The second path consisted of an extended dialogue/competition/argument between Hewitt (and his students) and Sussman (and his students), with both sides drawing in ideas from the rest of the world and spinning some off as well. This second path was characterized by a quest for "the right thing" where each new set of ideas was exemplified in the form of a new language, usually implemented on top of a Lisp-like language (MacLisp or Muddle) for the sake of rapid prototyping and experimentation.

The next round in the Hewitt/Sussman dialogue was, of course, Scheme (as discussed in section 2.8); in hindsight, we observe that this development seems to have ended the dialogue, perhaps because it brought the entire path of exploration full circle. Starting from Lisp, they sought to explicate issues of search, of control structures, of models of computation, and finally came back simply to good old Lisp, but with a difference: lexical scoping—closures, in short—were needed to make Lisp compatible with the lambda calculus not merely in syntax but also in semantics, thereby connecting it firmly with various developments in mathematical logic and paving the way for the Lisp community to interact with developments in functional programming.

(One consequence of this hookup is that it has for some years now become quite respectable to assume Scheme without further explanation as a framework for discussion at the annual ACM Principles of Programming Languages conference!) Hewitt had noted that the actor model could capture the salient aspects of the lambda calculus; Scheme demonstrated that the lambda calculus captured nearly all salient aspects (excepting only side effects and synchronization) of the actor model.

Sussman and Steele began to look fairly intensely at the semantics of Lisp-like as well as actor-based languages in this new light. Scheme was so much simpler even than Lisp 1.5, once one accepted the overheads of maintaining lexical environments and closures, that one could write a complete interpreter for it in Lisp on a single sheet of paper (or in two 30-line screenfuls). This allowed for extremely rapid experimentation with language and implementation ideas; at one point Sussman and Steele were testing and measuring as many as ten new interpreters a week. Some of their results were summarized in *The Art of the Interpreter* [Steele, 1978b]. A particular point of

interest was comparison of call-by-name and call-by-value parameters; in this they were influenced by work at Indiana University discussed in the paper *CONS Should Not Evaluate Its Arguments* [Friedman, November].

Besides being itself susceptible to rapid mutation, Scheme has also served as an implementation base for rapid prototyping of yet other languages. One popular technique among theoreticians for formally describing the meaning of a language is to give a denotational semantics, which describes the meaning of each construct in terms of its parts and their relationships; lambda calculus is the glue of this notation. Because Scheme in effect provides a simple and practical implementation of lambda calculus in a usable programming environment, a denotational semantics can be transcribed straightforwardly into Scheme to produce a working interpreter for the new language.

While Scheme showed that a properly designed Lisp gave one all the flexibility (if not all the syntax) one needed in managing control structure and message-passing, it did not solve the other goal of the development of Lisp-based AI languages: the automatic management of goal-directed search or of theorem proving. After Scheme, a few new Lisp-based languages were developed in this direction by Sussman and his students, including constraint-based systems [Sussman, 1975a; Stallman, 1976; Steele, 1979; Kleer, 1978b] and truth maintenance systems [Kleer, 1978a; McAllester, 1978] based on non-monotonic logic. The technique of dependency-directed backtracking eliminated the “giant nest of FORALL loops” effect of chronological backtracking. Over time this line of research became more of a database design problem than a language design problem, and has not yet resulted in feedback to the mainstream evolution of Lisp.

Development of languages for artificial intelligence applications continued at other sites, however, and Lisp has remained the vehicle of choice for implementing them. During the early 1980's C became the alternative of choice for a while, especially where efficiency was a major concern. Improvements in Lisp implementation techniques, particularly in compilation and garbage collection, have swung that particular pendulum back a bit.

During the AI boom of the early 1980's, “expert systems” was the buzzword; this was usually understood to mean rule-based systems written in languages superficially not very different from METEOR or CONVERT. OPS5 was one of the better-known rule-based languages of this period, and XCON (an expert system for configuring VAX installations, developed by Carnegie-Mellon University for Digital Equipment Corporation) was its premier application success story. OPS5 was first implemented in Lisp; later it was recoded for efficiency in BLISS [Wulf, 1971] (a CMU-developed and DEC-supported systems implementation language at about the same semantic level as C).

Another important category of AI languages was frame-based; a good example was KRL (Knowledge Representation Language), which was implemented in Interlisp.

Another line of experimentation in Lisp is in the area of parallelism. While early developments included facilities for interrupt handling and multiprogramming, true multiprocessing evolved only with the availability of appropriate hardware facilities (in some cases built for the purpose). S1 Lisp [Brooks, 1982a] was designed to use the multiple processors of an S1 system, but (like so many other features of S1 Lisp) that part never really worked. Some of the most important early “real” parallel Lisp implementations were Multilisp, Qlisp, and Butterfly PSL.

Multilisp [Halstead, 1984; Halstead, 1985] was the work of Bert Halstead and his students at MIT. Based on Scheme, it relied primarily on the notion of a *future*, which is a sort of laundry ticket, a promise to deliver a value later once it has been computed. Multilisp also provided a PCALL construct, essentially a function call that evaluates the arguments concurrently (and completely) before invoking the function. PCALL thus provides a certain structured discipline for the use of futures that is adequate for many purposes. Multilisp ran on the Concert multiprocessor, a collection of 32 Motorola 68000 processors. MultiScheme, a descendant of Multilisp, was later

implemented for the BBN Butterfly [Miller, 1987].

Butterfly PSL [Swanson, 1988] was an implementation of Portable Standard Lisp [Griss, 1982] on the BBN Butterfly. It also relied entirely on futures for the spawning of parallel processes.

Qlisp [Gabriel, 1984; Goldman, 1988] was developed by Richard Gabriel and John McCarthy at Stanford. It extended Common Lisp with a number of parallel control structures that parallel (pun intended) existing Common Lisp control constructs, notably QLET, QLAMBDA, and QCATCH. The computational model involved a global queue of processes and a means of spawning processes and controlling their interaction and resource consumption. For example, QLAMBDA could produce three kinds of functions: normal ones, as produced by LAMBDA; eager ones, which would spawn a separate process when created; and delayed ones, which would spawn a separate process when invoked. Qlisp was implemented on the Alliant FX8 and was the first compiled parallel Lisp implementation.

Connection Machine Lisp [Steele, 1986] was a dialect of Common Lisp extended with a new data structure, the *xapping* intended to support fine-grain data parallelism. A xapping was implementationally a strange hybrid of array, hash table, and association list; semantically it is a set of ordered index-value pairs. The primitives of the language are geared toward processing all the values of a xapping concurrently, matching up values from different xappings by their associated indexes. The idea was that indexes are labels for virtual processors. If the storage management system does its job, values with the same index will in fact reside in the same physical processor, and interprocessor communication will occur only as a consequence of interactions of values with different indexes. While Connection Machine Lisp was reasonably pleasant to program in—sort of symbolic version of APL with many of the representational advantages of Lisp—in practice the required storage management was very difficult to implement effectively. A prototype implementation by Skef Wholey (who had earlier worked on Spice Lisp under Scott Fahlman at CMU) for Connection Machine was used as a research tool at Thinking Machines for a while but never released as a product. It worked well for simple programs but the prototype storage allocator did not behave well when xappings were nested. It was left on the back burner when Wholey returned to CMU to complete his doctorate. Sabot's Paralation Lisp [Sabot, 1988] was designed in part as a reaction to the problems of controlling communication in Connection Machine Lisp. The primary contribution of Connection Machine Lisp was to work out, using a Scheme-like interpreter as an expository vehicle, some of the theoretical and pragmatic consequences of providing a functional language such as FP [Backus, 1978] with an explicitly parallel operational interpretation. Steele later explored the further consequences of nested parallelism and exhibited yet another Scheme-like interpreter that could detect race conditions in parallel programs [Steele, 1990a].

To recapitulate: Lisp is an excellent laboratory for language experimentation for two reasons. First, one can choose a very small subset, with only a dozen primitives or so, that is still recognizably a member of the class of Lisp-like languages. It is very easy to bootstrap such a small language, with variations of choice, on a new platform. If it looks promising, one can flesh out the long laundry list of amenities later. Second, it is particularly easy—the work of an hour or less—to bootstrap such a new dialect within an existing Lisp implementation. Even if the host implementation differs in fundamental ways from the new dialect, it can provide primitive operations such as arithmetic and I/O as well as being a programming language that is just plain convenient for writing language interpreters. If you can live with the generic, list-structure-oriented syntax, you can have a field day reprogramming the semantics. After you get that right there is time enough to re-engineer it and, if you must, slap a parser on the front.

5 Why Lisp is Diverse

In this history of the evolution of Lisp we have seen that Lisp seems to have a more elaborate and complex history than languages with wider usage. It would seem that almost every little research group has its own version of Lisp, and there would appear to be as many Lisps as variations on language concepts. It is natural to ask what is so special or different about Lisp that explains it.

There are five basic reasons: its theoretical foundations, its malleability, its interactive and incremental nature, its operating system facilities, and the people who choose to work on it.

Its theoretical foundations. Lisp was founded on the footing of recursive function theory and the theory of computability. Its purest form is useful for mathematical reasoning and proof. Therefore, many theoretically minded researchers have adopted Lisp or Lisp-like languages in which to express their ideas and to do their work. We thus see many Lisp-oriented papers with new language constructs explained, existing constructs explained, properties of programs proved, and proof techniques explored.

The upshot is that Lisp and Lisp-like languages are always in the forefront of basic language research. And it is common for more practically minded theoretical researchers to also implement their ideas in Lisp.

Its malleability. It is easy with Lisp to experiment with new language features, because it is possible to extend Lisp in such a way that the extensions are indistinguishable to users from the base language. Primarily this is accomplished through the use of macros, which have been part of Lisp since 1963 [Hart, 1963]. Lisp macros, with their use of Lisp as a computation engine to compute expansions, have proved to be a more effective way to extend a language than the string-processing mechanisms of other languages. Furthermore, such macro-based extensions are accepted within the Lisp community in a way that is not found in other language communities.

Furthermore, more recent Lisp dialects have provided mechanisms to extend the type system. This enables people to experiment with new data types. Of course, other languages have had this mechanism, but in Lisp the data typing mechanism combines with the powerful macro facility and the functional nature of the language to allow entirely new computing paradigms to be built in Lisp. For example, we have seen data-driven paradigms [Sussman, 1971], possible-worlds paradigms [McDermott, 1974], and object-oriented paradigms [Moon, 1986] [Bobrow, 1986] implemented in Lisp in such a way that the seams between Lisp and these new paradigms are essentially invisible.

Its interactive and incremental nature. It is easy to explore the solutions to programming problems in Lisp, because it is easy to implement part of a solution, test it, modify it, change design, and debug the changes. There is no lengthy edit-compile-link cycle. Because of this, Lisp is useful for rapid prototyping and for constructing very large programs in the face of an incomplete—and possibly impossible to complete—plan of attack. Therefore, Lisp has often been used for exploring territory that is too imposing with other languages.

This characteristic of Lisp makes it attractive to the adventuresome and pioneering.

Its operating system facilities. Many Lisp implementations provide facilities reminiscent of operating systems: a command processor, an automatic storage management facility, file management, display (windows, graphics, mouse) facilities, multitasking, a compiler, an incremental (re)linker/loader, a symbolic debugger, performance monitoring, and sometimes multiprocessing (parallel programming).

This means that it is possible to do operating system research in Lisp and to provide a complete operating environment. Combined with its interactive and incremental nature, it is possible to write sophisticated text editors, and to supplant the native operating system of the host computer. Thus, a Lisp system can provide an operating environment that provides strong portability across a wide variety of incompatible platforms.

This makes Lisp an attractive vehicle for researchers and thereby further diversifies Lisp.

Its people. Of course, languages do not diversify themselves, people diversify languages. The above four factors merely serve to attract people to Lisp, and they provide facilities for them to experiment with Lisp. If the people attracted to Lisp were not interested in exploring new language alternatives, then Lisp would not have been diversified, so there must be something about Lisp that attracts adventuresome people.

Lisp is the language of artificial intelligence, among other things. And AI is a branch of computer science that is directed towards exploring the most difficult and exotic of all programming tasks: mimicking or understanding cognition and intelligence. The people who are attracted to AI are generally creative and bold, and the language designers and implementors follow in this mold, often being AI researchers or former AI researchers themselves.

Lisp provides its peculiar set of characteristics because those features—or ones like them—were required for the early advances of AI. Only when AI was the subject of commercial concerns did AI companies turn to other languages than Lisp.

Another attraction is that Lisp is a language of experts, which for our purposes means that Lisp is not a language designed for inexperienced programmers to code robust reliable software. Therefore, there is little compile-time type checking, there are few module systems, there is little safety or discipline built into the language. It is an anarchic language, while most other languages are “fascist” (as hackers would have it [Raymond, 1991]).

The key point is that a certain sort of person is attracted to Lisp, and that person is interested in programming a certain way, in exploring new language designs, and in demonstrating or exercising creativity.

6 The Great Consolidation

A phenomenon that we would have liked to have explored but didn’t was the great consolidation of the ’80’s. Looking at the overall history of programming languages, it seems clear that during the 1960’s and 1970’s there were a profusion of programming languages developed, many of them by very small isolated groups. This fits our model of diversification—diversification happens when there is no sufficiently large and powerful acceptance group and when language developers are isolated.

But in the 1980’s, almost all language development was consolidation—the ’80’s were boring.

There are some possible explanations for this, each concerning a possible acceptance group, and, of course, the explanations combine. One explanation is that the network—the ARPANet in particular—solidified and was accessible to many more people. This created a large acceptance group and a coincident push for portability and standardization.

A second explanation is that the 1980’s saw the solidification of the PC marketplace, which provides a commercial acceptance group and, again, a push for portability.

A third explanation is that the workstation marketplace, which is both a commercial acceptance group and a large network-based, user-oriented acceptance group, began to develop and drained off a lot of diversification talent in pursuit of fame and fortune.

A variation on this explanation is that commercial languages are best when they provide guidance and discipline to help the not-so-talented programmer to develop robust, reliable, maintainable software, and such languages are just not very interesting for diversifiers in the 1980’s, given the well-entrenched techniques and practices developed in the ’60’s and ’70’s for achieving those goals.

A fourth explanation is that designing complete languages rather than exploring individual concepts or features became too hard to do or too unfashionable. In the ’60’s and ’70’s, it was

possible for a single person or a small group to design and implement a language with sufficient performance to be useful. In the 1980's, because commercial pressure forced performance improvements in commercially supported languages, it became too much of a task to produce a language that would even have an acceptance group, let alone one large or powerful enough to enter the language into a consolidation phase.

Designing and implementing languages in the 1980's is perhaps too difficult a means to achieve tenure or fame or a promotion, and so the tendency is to abandon the activity.

Here are how some others have put it:

LISP is unusual, in the sense that it clearly deviates from every other type of programming language that has ever been developed. . . . The theoretical concepts and implications of LISP far transcend its practical usage.

—Jean E. Sammet [?, p. 406]

This is one of the great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. . . . After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues.

—Abelson and Sussman[?, p. xvii]

Syntactic sugar causes cancer of the semicolon.

—Alan Perlis

What I like about Lisp is that you can feel the bits between your toes.

—Drew McDermott [?]

Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems.

—Alan Perlis (forward to [?])

Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms. . . .

—Alan Perlis (forward to [?])

Lisp is the medium of choice for people who enjoy free style and flexibility.

—Gerald Jay Sussman (intro to The Little Lisper, p. ix)

Hey, Quux: Let's quit hacking this paper and hack Lisp instead!

—rpg (the final edit)

7 Conclusions

[The following text is very sketchy and needs to be filled out.]

We're still working on this.

[End of sketchy section.]

References

- [ACM LFP, 1982] ACM SIGPLAN/SIGACT/SIGART. *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania, August 1982. Association for Computing Machinery. ISBN 0-89791-082-6.
- [ACM LFP, 1984] ACM SIGPLAN/SIGACT/SIGART. *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984. Association for Computing Machinery. ISBN 0-89791-142-3.
- [ACM LFP, 1986] ACM SIGPLAN/SIGACT/SIGART. *Proc. 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986. Association for Computing Machinery. ISBN 0-89791-200-4.
- [ACM LFP, 1988] ACM SIGPLAN/SIGACT/SIGART. *Proc. 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. Association for Computing Machinery. ISBN 0-89791-273-X.
- [ACM OOPSLA, 1986] *Proceedings of the ACM Conference on Objected-Oriented Programming, Systems, Languages, and Applications (OOPSLA '86)*, Portland, Oregon, October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21:11, November 1986. ISBN 0-89791-204-7.
- [Backus, 1978] Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:8, pp. 613–641, August 1978. 1977 ACM Turing Award Lecture.
- [Baker, 1978] Baker, Henry B., Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21:4, pp. 280–294, April 1978.
- [Bartley, 1986] Bartley, David H., and John C. Jensen. The implementation of PC Scheme. In [ACM LFP, 1986], pp. 86–93.
- [Bawden, 1988] Bawden, Alan, and Jonathan Rees. Syntactic closures. In [ACM LFP, 1988], pp. 86–95.
- [Berkeley, 1985] Berkeley, Edmund C., and Daniel G. Bobrow, eds. *The Programming Language LISP: Its Operation and Applications*. Information International, Inc. and MIT Press, Cambridge, Massachusetts, 1985.
- [Black, 1985] Black, Fischer. Styles of programming in LISP. In [Berkeley, 1985], pp. 96–107.
- [Bobrow, 1972] Bobrow, Robert J., Richard R. Burton, and Daryle Lewis. *UCI-LISP Manual (An Extended Stanford LISP 1.6 System)*. Information and Computer Science Technical Report 21, University of California, Irvine, Irvine, California, October 1972.
- [Bobrow, 1973] Bobrow, Daniel G., and Ben. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16:10, pp. 591–603, October 1973.
- [Bobrow, 1985] Bobrow, Daniel G. METEOR: A LISP interpreter for string transformations. In [Berkeley, 1985], pp. 161–190.
- [Bobrow, 1986] Bobrow, Daniel G., Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In [ACM OOPSLA, 1986], pp. 17–29.
- [Boehm, 1986] Boehm, Hans-J., Robert Cartwright, Mark Riggall, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In [ACM LFP, 1986], pp. 162–173.
- [Brooks, 1982a] Brooks, Rodney A., Richard P. Gabriel, and Guy L. Steele Jr. S-1 Common Lisp implementation. In [ACM LFP, 1982], pp. 108–113.
- [Brooks, 1982b] Brooks, Rodney A., Richard P. Gabriel, and J Steele, Guy L. An optimizing compiler for lexically scoped LISP. In *Proceedings of the 1982 Symposium on Compiler Construction*, pp. 261–275, Boston, June 1982. ACM SIGPLAN, Association for Computing Machinery. *ACM SIGPLAN Notices*, 17:6, June 1982. ISBN 0-89791-074-5.

- [Brooks, 1984] Brooks, Rodney A., and Richard P. Gabriel. A critique of Common Lisp. In [ACM LFP, 1984], pp. 1–8.
- [Burke, 1983] Burke, G. S., G. J. Carrette, and C. R. Eliot. *NIL Reference Manual*. Report MIT/LCS/TR-311, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1983.
- [Burstall, 1971] Burstall, R. M., J. S. Collins, and R. J. Popplestone, eds. *Programming in POP-2*. Edinburgh University Press, 1971.
- [Campbell, 1984] Campbell, J. A., ed. *Implementations of Prolog*. Ellis Horwood Limited, Chichester, 1984. ISBN 0-470-20045-6. Also published by John Wiley & Sons, New York.
- [Clark, 1982] Clark, K. L., and S.-Å. Tärnlund, eds. *Logic Programming*. Academic Press, New York, 1982.
- [Clinger, 1984] Clinger, William. The Scheme 311 compiler: An exercise in denotational semantics. In [ACM LFP, 1984], pp. 356–364.
- [Clinger, 1985a] Clinger, William (ed.). *The Revised Revised Report on Scheme; or, An Uncommon Lisp*. AI Memo 848, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1985.
- [Clinger, 1985b] Clinger, William (ed.). *The Revised Revised Report on Scheme; or, An Uncommon Lisp*. Computer Science Department Technical Report 174, Indiana University, Bloomington, June 1985.
- [Clinger, 1988] Clinger, William D., Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In [ACM LFP, 1988], pp. 124–131.
- [Clinger, 1990] Clinger, William D. How to read floating point numbers accurately. In *Proc. 1990 ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 92–101, White Plains, New York, June 1990. Association for Computing Machinery. Proceedings published as *ACM SIGPLAN Notices* **25**, 6 (June 1990).
- [Clinger, 1991] Clinger, William, and Jonathan Rees. Macros that work. In *Proc. Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 155–162, Orlando, Florida, January 1991. ACM SIGACT/SIGPLAN, Association for Computing Machinery. ISBN 0-89791-419-8.
- [Correll, 1979] Correll, Steven. S-1 uniprocessor architecture (sma-4). In *The S-1 Project 1979 Annual Report*, volume I, chapter 4. Lawrence Livermore Laboratory, Livermore, California, 1979.
- [Davies, 1984] Davies, J. POPLER: Implementation of a POP-2-based PLANNER. In [Campbell, 1984], pp. 28–49.
- [DEC, 1964] Digital Equipment Corporation, Maynard, Massachusetts. *Programmed Data Processor-6 Handbook*, 1964.
- [DEC, 1969] Digital Equipment Corporation, Maynard, Massachusetts. *PDP-10 Reference Handbook*, 1969.
- [DEC, 1981] Digital Equipment Corporation, Maynard, Massachusetts. *VAX Architecture Handbook*, 1981.
- [Deutsch, 1973] Deutsch, L. Peter. A LISP machine with very compact programs. In [IJCAI, 1973], pp. 697–703.
- [Deutsch, 1976] Deutsch, L. Peter, and Daniel G Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19:9, pp. 522–526, September 1976.
- [Deutsch, 1985] Deutsch, L. Peter, and Edmund C. Berkeley. The LISP implementation for the PDP-1 computer. In [Berkeley, 1985], pp. 326–375.
- [Drescher, 1987] Drescher, Gary. *ObjectLISP User Manual*. LMI (Lisp Machines Incorporated), Cambridge, Massachusetts, 1987.
- [Dybvig, 1986] Dybvig, R. Kent, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In [ACM LFP, 1986], pp. 143–150.
- [Eastlake, 1968] Eastlake, D., R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. *ITS 1.5 Reference Manual*. AI Memo 161, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1968. Revised as Memo 161A, July 1969.

- [Eastlake, 1972] Eastlake, Donald E. *ITS Status Report*. AI Memo 238, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, April 1972.
- [Fateman, 1973] Fateman, Richard J. Reply to an editorial. *ACM SIGSAM Bulletin*, 25, pp. 9–11, March 1973. This reports the results of a test in which a compiled MacLisp floating-point program was faster than equivalent Fortran code. The numerical portion of the code was identical and MacLisp used a faster subroutine-call protocol.
- [Feldman, 1972] Feldman, J. A., J. R. Low, D. C. Swinehart, and R. H. Taylor. Recent developments in SAIL. In *Proceedings of the 1972 AFIPS Fall Joint Computer Conference*, volume 41, pp. 1193–1202, Stanford, California, November 1972. American Federation of Information Processing Societies.
- [Fessenden, 1983] Fessenden, Carol, William Clinger, Daniel P. Friedman, and Christopher Haynes. *Scheme 311 Version 4 Reference Manual*. Technical Report 137, Indiana University, February 1983.
- [Foderaro, 1982] Foderaro, J. K., and K. L. Sklower. *The FRANZ Lisp Manual*. University of California, Berkeley, California, April 1982.
- [Forgy, 1977] Forgy, C., and J. McDermott. OPS, a domain-independent production system language. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 933–935, Cambridge, Massachusetts, August 1977. International Joint Council on Artificial Intelligence.
- [Friedman, November] Friedman, Daniel P., and David S. Wise. *Cons Should Not Evaluate Its Arguments*. Technical Report 44, Indiana University, 1975 November.
- [Gabriel, 1982] Gabriel, Richard P., and Larry M. Masinter. Performance of Lisp systems. In [ACM LFP, 1982], pp. 123–142.
- [Gabriel, 1984] Gabriel, Richard P., and John McCarthy. Queue-based multiprocessing Lisp. In [ACM LFP, 1984], pp. 25–44.
- [Gabriel, 1985] Gabriel, Richard P. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, Massachusetts, 1985. ISBN 0-262-07093-6.
- [Gabriel, 1988] Gabriel, Richard P., and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1:1, pp. 81–101, June 1988. ISSN 0892-4635.
- [Galley, 1975] Galley, S.W., and Greg. Pfister. *The MDL Language*. Programming Technology Division Document SYS.11.01, MIT Project MAC, Cambridge, Massachusetts, November 1975.
- [Geschke, 1977] Geschke, Charles M., J Morris, James H., and Edwin H. Satterthwaite. Early experience with Mesa. *Communications of the ACM*, 20:8, pp. 540–553, August 1977.
- [Golden, 1970] Golden, Jeffrey P. *A User's Guide to the A. I. Group LISCOP Lisp Compiler: Interim Report*. AI Memo 210, MIT Project MAC, Cambridge, Massachusetts, December 1970.
- [Goldman, 1988] Goldman, Ron, and Richard P. Gabriel. Preliminary results with the initial implementation of Qlisp. In [ACM LFP, 1988], pp. 143–152.
- [Greenblatt, 1974] Greenblatt, Richard. *The LISP Machine*. Working Paper 79, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, November 1974.
- [Greussay, 1977] Greussay, P. *Contribution à la définition interprétative et à l'implémentation des lambda-langages*. Thèse d'Etat, Université de Paris VI, November 1977.
- [Griss, 1981] Griss, Martin L., and Anthony C. Hearn. A portable LISP compiler. *Software Practice and Experience*, 11, pp. 541–605, 1981.
- [Griss, 1982] Griss, Martin L., Eric Benson, and Gerald Q. Maguire J. PSL: A portable LISP system. In [ACM LFP, 1982], pp. 88–97.
- [Group, 1982] Group, Utah Symbolic Computation. *The Portable Standard LISP Users Manual*. Technical Report TR-10, Department of Computer Science, University of Utah, Salt Lake City, JAN 1982.

- [Guzman, 1966] Guzman, Adolfo, and Harold V. McIntosh. *CONVERT*. AI Memo 99, MIT Project MAC, Cambridge, Massachusetts, June 1966.
- [Hailpern, 1979] Hailpern, Brent T., and Bruce L. Hitson. *S-1 Architecture Manual*. Technical Report 161 (STAN-CS-79-715), Department of Electrical Engineering, Stanford University, Stanford, California, January 1979.
- [Halstead, 1984] Halstead, Robert H., Jr. Implementation of Multilisp: Lisp on a multiprocessor. In [ACM LFP, 1984], pp. 9–17.
- [Halstead, 1985] Halstead, Robert H., Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:4, pp. 501–538, October 1985.
- [Harbison, 1991] Harbison, Samuel P., and J Steele, Guy L. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, third edition, 1991. ISBN 0-13-110933-2.
- [Hart, 1963] Hart, Timothy P. *MACRO Definitions for LISP*. AI Memo 57, MIT Artificial Intelligence Project—RLE and MIT Computation Center, Cambridge, Massachusetts, October 1963.
- [Hart, 1985] Hart, Timothy P., and Thomas G. Evans. Notes on implementing LISP for the M-460 computer. In [Berkeley, 1985], pp. 191–203.
- [Hearn, 1971] Hearn, A. C. REDUCE 2: A system and language for algebraic manipulation. In *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, pp. 128–133, Los Angeles, March 1971.
- [Henneman, 1985] Henneman, William. An auxiliary language for more natural expression—The A-language. In [Berkeley, 1985], pp. 239–248.
- [Hewitt, 1969] Hewitt, Carl. PLANNER: A language for proving theorems in robots. In *Proceedings of the [First] International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 295–301, Washington, D. C., May 1969. International Joint Council on Artificial Intelligence.
- [Hewitt, 1972] Hewitt, Carl. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1972. MIT Artificial Intelligence Laboratory TR-258.
- [Hewitt, 1975] Hewitt, Carl. *How to Use What You Know*. Working Paper 93, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, May 1975.
- [Hieb, 1990] Hieb, Robert, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. 1990 ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 66–77, White Plains, New York, June 1990. Association for Computing Machinery. Proceedings published as *ACM SIGPLAN Notices* **25**, 6 (June 1990).
- [IEEE, 1985] IEEE, New York. *IEEE Standard for Binary Floating-Point Arithmetic*, ansi/ieee std 754-1985 edition, 1985. An American National Standard.
- [IEEE, 1991] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, ieee std 1178-1990 edition, 1991.
- [IJCAI, 1973] International Joint Council on Artificial Intelligence. *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI3)*, Stanford, California, August 1973.
- [Iverson, 1962] Iverson, Kenneth E. *A Programming Language*. Wiley, New York, 1962.
- [Jensen, 1974] Jensen, Kathleen, and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, 1974.
- [Kahn, 1984] Kahn, K. M., and M. Carlsson. How to implement Prolog on a LISP machine. In [Campbell, 1984], pp. 117–134.
- [Kempf, 1987] Kempf, James, Warren Harris, Roy D'Souza, and Alan Snyder. Experience with Common-Loops. In *Proceedings of the ACM Conference on Objected-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)*, pp. 214–226, Orlando, Florida, October 1987. Association for Computing Machinery. *ACM SIGPLAN Notices*, 22:12, December 1987. ISBN 0-89791-247-0.

- [Kernighan, 1978] Kernighan, Brian W., and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [Kleer, 1978a] Kleer, Johan de, Jon Doyle, Charles Rich, J Steele, Guy L., and Gerald Jay Sussman. *AMORD: A Deductive Procedure System*. AI Memo 435, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1978.
- [Kleer, 1978b] Kleer, Johan de, and Gerald Jay Sussman. *Propagation of Constraints Applied to Circuit Synthesis*. AI Memo 485, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, September 1978. Also in *i[CircuitTheoryandApplications]b[8]* (1980), 127-144.
- [Knuth, 1969] Knuth, Donald E. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1969.
- [Knuth, 1974] Knuth, Donald E. Structured programming with GO TO statements. *Computing Surveys*, 6:4, pp. 261–301, December 1974.
- [Knuth, 1981] Knuth, Donald E. *Seminumerical Algorithms (Second Edition)*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1981.
- [Knuth, 1986] Knuth, Donald E. *The METAFONT Book*, volume C of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1986.
- [Kohlbecker, 1986a] Kohlbecker, Eugene, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In [ACM LFP, 1986], pp. 151–161.
- [Kohlbecker, 1986b] Kohlbecker, Jr., Eugene E. *Syntactic Extensions in the Programming Language Lisp*. Technical Report 109, Indiana University, August 1986. Ph.D. thesis.
- [Komorowski, 1982] Komorowski, H. J. QLOG: The programming environment for PROLOG in LISP. In [Clark, 1982], pp. 315–322.
- [Kranz, 1986] Kranz, David, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proc. 1986 ACM SIGPLAN '86 Symposium on Compiler Construction*, pp. 219–233, Palo Alto, California, June 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21:7, July 1986. ISBN 0-89791-197-0.
- [Landin, 1964] Landin, Peter J. The mechanical evaluation of expressions. *Computer Journal*, 6:4, 1964.
- [Landin, 1965] Landin, Peter J. A correspondence between ALGOL 60 and Church's lambda-notation. *Communications of the ACM*, 8:2-3, February–March 1965.
- [Lisp Conference, 1980] Republished by Association for Computing Machinery. *Conference Record of the 1980 LISP Conference*, Stanford, California, August 1980.
- [Malachi, 1984] Malachi, Yonathan, Zohar Manna, and Richard Waldinger. TABLOG: The deductive-tableau programming language. In [ACM LFP, 1984], pp. 323–330.
- [Marti, 1979] Marti, J., A. C. Hearn, M. L. Griss, and C. Griss. Standard lisp report. *ACM SIGPLAN Notices*, 14:10, pp. 48–68, October 1979.
- [Mathlab Group, 1977] Mathlab Group, The. *MACSYMA Reference Manual (Version Nine)*. MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1977.
- [McAllester, 1978] McAllester, David A. *A Three Valued Truth Maintenance System*. AI Memo 473, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, May 1978.
- [McCarthy, 1962] McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, , and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962.
- [McCarthy, 1980] McCarthy, John. Lisp: Notes on its past and future. In [Lisp Conference, 1980], pp. v–viii.
- [McCarthy, 1981] McCarthy, John. History of LISP. In Wexelblat, Richard L., ed., *History of Programming Languages*, ACM Monograph Series, chapter IV, pp. 173–197. Academic Press, New York, 1981. ISBN 0-12-745040-8.

- [McDermott, 1974] McDermott, Drew V., and Gerald Jay Sussman. *The CONNIVER Reference Manual*. AI Memo 295a, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1974.
- [McDermott, 1980] McDermott, Drew. An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP. In [Lisp Conference, 1980], pp. 154–162.
- [Mellish, 1984] Mellish, C., and S. Hardy. Integrating Prolog in the POPLOG environment. In [Campbell, 1984], pp. 147–162.
- [Miller, 1987] Miller, James Slocum. *MultiScheme: A Parallel Processing System Mased on MIT Scheme*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1987.
- [MIT RLE, 1962a] MIT Research Laboratory of Electronics. MIT Press, Cambridge, Massachusetts. *COMIT Programmers Reference Manual*, June 1962.
- [MIT RLE, 1962b] MIT Research Laboratory of Electronics. MIT Press, Cambridge, Massachusetts. *An Introduction to COMIT Programming*, June 1962.
- [Moon, April 1974] Moon, David A. *MacLISP Reference Manual*. MIT Project MAC, Cambridge, Massachusetts, April 1974.
- [Moon, 1984] Moon, David A. Garbage collection in a large Lisp system. In [ACM LFP, 1984], pp. 235–246.
- [Moon, 1986] Moon, David A. Object-oriented programming with flavors. In [ACM OOPSLA, 1986], pp. 1–8.
- [Moore, 1976] Moore, J. Strother II. *The InterLISP Virtual Machine Specification*. Technical Report CSL 76-5, Xerox Palo Alto Research Center, Palo Alto, California, September 1976.
- [Naur, 1963] Naur, Peter (ed.), et al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1, pp. 1–20, January 1963.
- [Okuno, 1984] Okuno, Hiroshi G., Ikuo Takeuchi, Nobuyasu Osato, Yasushi Hibino, and Kazufumi Watanabe. TAO: A fast interpreter-centered Lisp system on Lisp machine ELIS. In [ACM LFP, 1984], pp. 140–149.
- [Organick, 1972] Organick, Elliot I. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, Massachusetts, 1972.
- [Padget, 1986] Padget, Julian, et al. Desiderata for the standardisation of Lisp. In [ACM LFP, 1986], pp. 54–66.
- [PDP-6 Lisp, 1967] *PDP-6 LISP (LISP 1.6)*. AI Memo 116, MIT Project MAC, Cambridge, Massachusetts, January 1967. Revised as Memo 116A, April 1967. The report does not bear the author’s name, but Jeffrey P. Golden [Golden, 1970] attributes it to Jon L White.
- [Pitman, 1980] Pitman, Kent M. Special forms in Lisp. In [Lisp Conference, 1980], pp. 179–187.
- [Pitman, 1983] Pitman, Kent M. *The Revised MacLISP Manual*. MIT/LCS/TR 295, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1983.
- [Pratt, 1973] Pratt, Vaughan R. Top down operator precedence. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 41–51, Boston, October 1973. ACM SIGACT and SIGPLAN, Association for Computing Machinery.
- [Pratt, 1976] Pratt, Vaughan R. *CGOL: An Alternative External Representation for LISP Users*. AI Working Paper 121, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976.
- [Quam, 1972] Quam, Lynn H., and Whitfield Diffie. *Stanford LISP 1.6 Manual*. SAIL Operating Note 28.6, Stanford Artificial Intelligence Laboratory, Stanford, California, 1972.
- [Raymond, 1991] Raymond, Eric, ed. *The New Hacker’s Dictionary*. MIT Press, Cambridge, Massachusetts, 1991.
- [Rees, 1982] Rees, Jonathan A., and Norman I. Adams IV. T: A dialect of Lisp; or, LAMBDA: The ultimate software tool. In [ACM LFP, 1982], pp. 114–122.

- [Rees, 1986] Rees, Jonathan, William Clinger, et al. The revised³ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21:12, pp. 37–79, December 1986.
- [Reynolds, 1972] Reynolds, John C. Definitional interpreters for higher order programming languages. In *Proc. ACM National Conference*, pp. 717–740, Boston, August 1972. Association for Computing Machinery.
- [Robinson, 1982] Robinson, J. A., and E. E. Sibert. LOGLISP: Motivation, design, and implementation. In [Clark, 1982], pp. 299–313.
- [Roylance, 1988] Roylance, Gerald. Expressing mathematical subroutines constructively. In [ACM LFP, 1988], pp. 8–13.
- [Rudloe, 1962] Rudloe, H. *Tape Editor*. Program Write-up BBN-101, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, January 1962.
- [Sabot, 1988] Sabot, Gary W. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, Massachusetts, 1988. ISBN 0-262-19277-2.
- [Saunders, 1985a] Saunders, Robert A. The LISP listing for the Q-32 compiler, and some samples. In [Berkeley, 1985], pp. 290–317.
- [Saunders, 1985b] Saunders, Robert A. The LISP system for the Q-32 computer. In [Berkeley, 1985], pp. 220–238.
- [Shaw, 1981] Shaw, Mary, Wm. A. Wulf, and Ralph L. London. Abstraction and verification in Alphard: Iteration and generators. In Shaw, Mary, ed., *ALPHARD: Form and Content*, chapter 3, pp. 73–116. Springer-Verlag, New York, 1981. ISBN 0-387-90663-0.
- [Smith, 1970] Smith, David Canfield. *MLISP*. Technical Report AIM-135, Stanford Artificial Intelligence Project, October 1970.
- [Sobalvarro, 1988] Sobalvarro, Patrick G. A Lifetime-based Gabrbage Collector for LISP Systems on General-Purpose Computers. Bachelor’s Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1988.
- [Smith, 1973] Smith, David Canfield, and Horace J. Enea. Backtracking in MLISP2: An efficient backtracking method for LISP. In [IJCAI, 1973], pp. 677–685.
- [Stallman, 1976] Stallman, Richard M., and Gerald Jay Sussman. *Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis*. AI Memo 380, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, September 1976. Also in *i[ArtificialIntelligence]b[9]* (1977), 135–196.
- [Steele, 1976a] Steele, Guy Lewis, Jr. *LAMBDA: The Ultimate Declarative*. AI Memo 379, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, November 1976.
- [Steele, 1976b] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. *LAMBDA: The Ultimate Imperative*. AI Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976.
- [Steele, 1977a] Steele, Guy L., Jr. Macaroni is better than spaghetti. In *Proceedings of the Artificial Intelligence and Programming Languages Conference*, pp. 60–66, Rochester, New York, August 1977. ACM SIGPLAN and SIGART. Proceedings published as *ACM SIGPLAN Notices* **12**, 8 (August 1977) and as *ACM SIGART Newsletter* **64** (August 1977).
- [Steele, 1977b] Steele, Guy Lewis, Jr. Compiler optimization based on viewing LAMBDA as rename plus goto. Master’s thesis, Massachusetts Institute of Technology, May 1977. Published as [Steele, 1978a].
- [Steele, 1977c] Steele, Guy Lewis, Jr. Data representations in PDP-10 MacLISP. In *Proceedings of the 1977 MACSYMA Users’ Conference*, pp. 203–214, Washington, D.C., July 1977. NASA Scientific and Technical Information Office.

- [Steele, 1977d] Steele, Guy Lewis, Jr. Debunking the ‘expensive procedure call’ myth; or, Procedure call implementations considered harmful; or, LAMBDA: The ultimate GOTO. In *Proceedings of the ACM National Conference*, pp. 153–162, Seattle, October 1977. Association for Computing Machinery.
- [Steele, 1977e] Steele, Guy Lewis, Jr. Fast arithmetic in macisp. In *Proceedings of the 1977 MACSYMA Users’ Conference*, pp. 215–224, Washington, D.C., July 1977. NASA Scientific and Technical Information Office.
- [Steele, 1978a] Steele, Guy Lewis, Jr. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)*. Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978. This is a revised version of the author’s master’s thesis [Steele, 1977b].
- [Steele, 1978b] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two)*. AI Memo 453, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, May 1978.
- [Steele, 1978c] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. *The Revised Report on SCHEME: A Dialect of LISP*. AI Memo 452, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1978.
- [Steele, 1979] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. Constraints. In *Proceedings of the APL 79 Conference*, pp. 208–225, Rochester, New York, June 1979. ACM SIGPLAN STAPL. *APL Quote Quad*, 9:4, June 1979.
- [Steele, 1980] Steele, Guy Lewis, Jr., and Gerald Jay Sussman. The dream of a lifetime: A lazy variable extent mechanism. In [Lisp Conference, 1980], pp. 163–172.
- [Steele, 1982] Steele, Jr., Guy L. An overview of Common Lisp. In [ACM LFP, 1982], pp. 98–107.
- [Steele, 1984] Steele, Guy L., Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common Lisp: The Language*. Digital Press, Burlington, Massachusetts, 1984.
- [Steele, 1986] Steele, Guy L., Jr., and W. Daniel Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. In [ACM LFP, 1986], pp. 279–297.
- [Steele, 1990a] Steele, Jr., Guy L. Making asynchronous parallelism safe for the world. In *Proc. Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 218–231, San Francisco, January 1990. ACM SIGACT/SIGPLAN, Association for Computing Machinery. ISBN 0-89791-343-4.
- [Steele, 1990b] Steele, Jr., Guy L., and Jon L White. How to print floating-point numbers accurately. In *Proc. 1990 ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation*, pp. 112–126, White Plains, New York, June 1990. Association for Computing Machinery. Proceedings published as *ACM SIGPLAN Notices* **25**, 6 (June 1990).
- [Steele, 1990c] Steele, Guy L., Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L White. *Common Lisp: The Language (Second Edition)*. Digital Press, Bedford, Massachusetts, 1990.
- [Sussman, 1971] Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. *Micro-PLANNER Reference Manual*. AI Memo 203A, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1971.
- [Sussman, 1972a] Sussman, Gerald Jay, and Drew Vincent McDermott. From PLANNER to CONNIVER—A genetic approach. In *Proc. 1972 Fall Joint Computer Conference*, pp. 1171–1179, Montvale, New Jersey, August 1972. AFIPS Press. This is the published version of *Why Conniving is Better than Planning*.
- [Sussman, 1972b] Sussman, Gerald Jay, and Drew Vincent McDermott. *Why Conniving is Better than Planning*. AI Memo 255A, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, April 1972.

- [Sussman, 1975a] Sussman, Gerald Jay, and Richard M Stallman. *Heuristic Techniques in Computer-Aided Circuit Analysis*. AI Memo 328, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1975.
- [Sussman, 1975b] Sussman, Gerald Jay, and J Steele, Guy Lewis. *SCHEME: An Interpreter for Extended Lambda Calculus*. AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1975.
- [Sussman, 1988] Sussman, Gerald Jay, and Matthew Halfant. Abstraction in numerical methods. In [ACM LFP, 1988], pp. 1–7.
- [Swanson, 1988] Swanson, Mark R., Robert R. Kessler, and Gary Lindstrom. An implementation of Portable Standard Lisp on the BBN Butterfly. In [ACM LFP, 1988], pp. 132–142.
- [Swinehart, 1972] Swinehart, D. C., and R. F. Sproull. *SAIL*. SAIL Operating Note 57.2, Stanford Artificial Intelligence Laboratory, Stanford, California, 1972.
- [Symbolics, 1985] Symbolics, Inc., Cambridge, Massachusetts. *Reference Guide to Symbolics-Lisp*, March 1985.
- [Takeuchi, 1983] Takeuchi, Ikuo, Hirochi Okuno, and Nobuyasu Ohsato. TAO: A harmonic mean of Lisp, Prolog, and Smalltalk. *ACM SIGPLAN Notices*, 18:7, pp. 65–74, July 1983.
- [Teitelman, 1966] Teitelman, Warren. *PILOT: A Step toward Man-Computer Symbiosis*. Technical Report MAC-TR-32, MIT Project MAC, September 1966. Ph.D. thesis.
- [Teitelman, 1971] Teitelman, W., D. G. Bobrow, A. K. Hartley, and D. L. Murphy. *BBN-LISP: TENEX Reference Manual*. Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1971.
- [Teitelman, 1973] Teitelman, Warren. CLISP: Conversational LISP. In [IJCAI, 1973], pp. 686–690.
- [Teitelman, 1974] Teitelman, Warren, et al. *InterLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, California, 1974. First revision.
- [Teitelman, 1978] Teitelman, Warren, et al. *InterLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, California, October 1978. Third revision.
- [Tesler, 1973] Tesler, Lawrence G., Horace J. Enea, and David C. Smith. The LISP70 pattern matching system. In [IJCAI, 1973], pp. 671–676.
- [Thacker, 1982] Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. In Siewiorek, Daniel P., C. Gordon Bell, and Allen Newell, eds., *Computer Structures: Principles and Examples*, Computer Science Series, chapter 33, pp. 549–572. McGraw-Hill, New York, 1982. ISBN 0-07-057302-6.
- [Travis, 1977] Travis, Larry, Masahiro Honda, Richard LeBlanc, and Stephen Zeigler. Design rationale for TELOS, a PASCAL-based AI language. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, pp. 67–76, Rochester, New York, August 1977. Association for Computing Machinery.
- [Vuillemin, 1988] Vuillemin, Jean. Exact real computer arithmetic with continued fractions. In [ACM LFP, 1988], pp. 14–27.
- [Wand, 1977] Wand, Mitchell, and Daniel P. Friedman. *Compiling Lambda Expressions Using Continuations and Factorization*. Technical Report 55, Indiana University, July 1977.
- [Waters, 1984] Waters, Richard C. Expressional loops. In *Proc. Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 1–10, Salt Lake City, Utah, January 1984. ACM SIGACT/SIGPLAN, Association for Computing Machinery. ISBN 0-89791-125-3.
- [Waters, 1989a] Waters, Richard C. *Optimization of Series Expressions, Part I: User's Manual for the Series Macro Package*. AI Memo 1082, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1989.

- [Waters, 1989b] Waters, Richard C. *Optimization of Series Expressions, Part II: Overview of the Theory and Implementation*. AI Memo 1083, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1989.
- [Wegbreit, 1970] Wegbreit, Ben. *Studies in Extensible Programming Languages*. PhD thesis, Harvard University, Cambridge, Massachusetts, 1970.
- [Wegbreit, 1971] Wegbreit, Ben. The ECL programming system. In *Proc. 1971 Fall Joint Computer Conference*, pp. 253–262, Montvale, New Jersey, August 1971. AFIPS Press.
- [Wegbreit, 1972] Wegbreit, Ben, Ben Brosgol, Glenn Holloway, Charles Prenner, and Jay Spitzen. *ECL Programmer's Manual*. Technical Report 21-72, Harvard University Center for Research in Computing Technology, Cambridge, Massachusetts, September 1972.
- [Wegbreit, 1974] Wegbreit, Ben, Glenn Holloway, Jay Spitzen, and Judy Townley. *ECL Programmer's Manual*. Technical Report 23-74, Harvard University Center for Research in Computing Technology, Cambridge, Massachusetts, December 1974.
- [Weinreb, November 1978] Weinreb, Daniel, and David Moon. *LISP Machine Manual, Preliminary Version*. MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, November 1978.
- [Weinreb, March 1981] Weinreb, Daniel, and David Moon. *LISP Machine Manual, Third Edition*. MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1981.
- [White, 1980] White, Jon L. Address/memory management for a gigantic LISP environment; or, GC considered harmful. In [Lisp Conference, 1980], pp. 119–127.
- [White, 1969–1982] White, Jon L, et al. *LISP ARCHIV*. On-line archive of MacLisp release notes., 1969–1982.
- [White, 1986] White, Jon L. Reconfigurable, retargetable bignums: A case study in efficient, portable Lisp system building. In [ACM LFP, 1986], pp. 174–191.
- [Wulf, 1971] Wulf, W.A., D.B. Russell, and A.N. Habermann. Bliss: A language for systems programming. *Communications of the ACM*, 14:12, pp. 780–790, December 1971.
- [Wulf, 1975] Wulf, William, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler*, volume 2 of *Programming Language Series*. American Elsevier, New York, 1975. ISBN 0-444-00164-6.
- [Yngve, 1972] Yngve, Victor H. *Computer Programming with COMIT II*. MIT Press, Reading, Massachusetts, 1972. ISBN 0-262-74007-9.