THE EVOLUTION OF LISP

Guy L. Steele Jr.
Thinking Machines Corporation

Richard P. Gabriel
Lucid, Inc.

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 some notable successes and 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.

CONTENTS
6.1 Introduction
6.2 Implementation Projects Chronology
6.3 Evolution of Some Specific Language Features
6.4 Lisp as a Language Laboratory
6.5 Why Lisp is Diverse
References
6.1 INTRODUCTION

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.

Moreover, we have had to omit a great deal of material for lack of space. The choice of topics presented here necessarily reflects our own experiences and biases. We apologize if your favorite corner of the sprawling Lisp community has gone unmentioned.

Section 6.2 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 6.3 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 6.4 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 6.5 draws some conclusions about why Lisp evolved as it did.

6.2 IMPLEMENTATION PROJECTS CHRONOLOGY

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 begin our story where McCarthy left off.

6.2.1 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 and the implementation straightforward. There were very few changes made to the original language.

In the early 1960s, Timothy P. Hart and Thomas G. 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 1964].

Robert Saunders and his colleagues at System Development Corporation implemented Lisp 1.5 on the IBM-built AN/FSQ-32/V computer, often called simply the Q-32 [Saunders 1964b]. 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 1964]. 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 Timesharing System (ITS) [Eastlake 1968, 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.

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 a local user group, which usually consisted of only 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, partly because of the pioneering feeling that each laboratory embodied.

An exception to all this was the Lisp 2 project [Abrahams 1966], a concerted language development effort that was funded by ARPA and represented a radical departure from Lisp 1.5. There appears to have been some hope that this design would supersede Lisp 1.5 and bring symbolic processing closer to ALGOL 60. (See Section 6.3.5 for an example of Lisp 2 code.) Lisp 2 was implemented for the Q-32 computer but never achieved wide acceptance. Jean Sammet remarked [Sammet 1969, p. 596]:

... in contrast to most languages, in which the language is first designed and then implemented... it was facetiously said of LISP 2 that it was “an implementation in search of a language.”

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

6.2.2 MacLisp

MacLisp was the primary Lisp dialect at the MIT AI Lab from the late 1960s until the early 1980s. 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].
6.2.2.1 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.

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 concatenate the lists (BOAT AIRPLANE SKATEBOARD) and (CAR TRUCK) in MacLisp, one would type this expression to EVAL:

```
(APPEND (QUOTE (BOAT AIRPLANE SKATEBOARD)) (QUOTE (CAR TRUCK)))
```

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

```
(APPEND ' (BOAT AIRPLANE SKATEBOARD) ' (CAR TRUCK))
```

The result would of course be (BOAT AIRPLANE SKATEBOARD CAR TRUCK).

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

```
APPEND ( (BOAT AIRPLANE SKATEBOARD) (CAR TRUCK) )
```

The first expression denotes a function, and the second is a list of arguments. 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. BBN-Lisp (and thus Interlisp) accommodated both: if the first input line contained a complete form and at least one character of a second form, then BBN-Lisp finished reading the second form and used the EVALQUOTE interface for that interaction; otherwise it read exactly one form and used the EVAL interface for that interaction.

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

```
APPEND ((BOAT AIRPLANE SKATEBOARD) (CAR TRUCK))
```

to one like

```
(APPEND (QUOTE (BOAT AIRPLANE SKATEBOARD)) (QUOTE (CAR TRUCK)))
```

before evaluation is performed. A similar confusion 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].

McCarty [McCarty 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

```
APPEND ((BOAT AIRPLANE SKATEBOARD) (CAR TRUCK))
```

be regarded not as a syntactically mutated version of

```
(APPEND (QUOTE (BOAT AIRPLANE SKATEBOARD)) (QUOTE (CAR TRUCK)))
```

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, however, that MacLisp's APPLY function supported this same kluge.)

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:

```lisp
(DEFUN LIST N
  (DO ((J N (- J 1))
      (ANSWER () (CONS (ARG J) ANSWER))
    ( (ZEROP J) ANSWER)))
```

Parameter N is bound to the number of arguments passed. The expression `(ARG J)` refers to the
jth 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)))`.

The simple but powerful macro facility on which DEFMACRO is based was introduced in MacLisp
in the mid-1960s. See Section 6.3.3.

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

```lisp
(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 nonlocal 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 [Lisp Archive 1972, item 2] so that ERRSET could be reserved for its
intended use of error trapping.

The lesson of ERRSET and CATCH is important. The designers of ERRSET and later ERR had in
mind a particular situation and defined a pair of primitives to address that situation. But because these
facilities provided a combination of useful and powerful capabilities (error trapping plus dynamic
nonlocal exits), programmers began to use these facilities in unintended ways. Then the designers had
to go back and split the desired functionality into pieces with alternative interfaces. This pattern of
careful design, unintended use, and later redesign is common in the evolution of Lisp.

LISP SESSION

237
The next phase of MacLisp development began when the developers of MacLisp started to see a large and influential user 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.

6.2.2.2 Later MacLisp

During the latter part of its life cycle, 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 1970s when the techniques in the prototype “fast arithmetic compiler” LISCOM [Golden 1970] were incorporated into the MacLisp compiler by Jon L White, who had already been the principal MacLisp maintainer and developer for several years. (John Lyle White was commonly known by his login name JONL, which can be pronounced as either “jönnell” (to rhyme with “O’Donnell”) or “john-ell” (two equally stressed syllables). Because of this, he took to writing his name as “Jon L White” rather than “John L. White”.)

Steele, who at age 17 had already hung out around MIT for several years and had implemented a Lisp system for the IBM 1130, was hired as a Lisp hacker in July 1972 by the MIT Mathlab group, which was headed by Joel Moses. Steele soon took responsibility for maintaining the MacLisp interpreter and run-time system, allowing Jon L to concentrate almost full time on compiler improvements.

The resulting new MacLisp compiler, NCOMPLR [Moon 1974; Lisp Archive; Pitman 1983], became 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, which 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].

Bignums—arbitrary precision integer arithmetic—were added circa 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 1974].

Richard Greenblatt started the MIT Lisp Machine project in 1974 [Greenblatt 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 Muddle (later called 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 Mathlab 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.

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 6.3.5.1 for further discussion of some of these experiments.

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 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) → nil and (cdr nil) → nil, and Interlisp adopted the concept of a read table.

By the mid-1970s 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 ten years of strong use and acceptance within its somewhat small but very influential user community. Its implementation strategy of a large assembly language core proved 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 resulted 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 MacLisp user community 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.

6.2.3 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 Ph.D. 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 1964], an English-like or ALGOL-like surface syntax for Lisp (see Section 6.3.5.1), 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 1964].)
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 fit in well with the work philosophy of Interlisp. The Interlisp model was to emulate an infinite login session. In Interlisp, the programmer worked with source code presented by a structure editor, which operated on source code in the form of memory-resident Lisp data structures. Any changes to the code were saved in a file, which served as a persistent repository for the programmer’s code. DWIM’s changes were saved also. The memory-resident structure was considered the primary representation of the program; the file was merely a stable backup copy. (The MacLisp model, in contrast, was for the programmer to work with ASCII files that represented the program, using a character-oriented editor. Here the file was considered the primary representation of the program.)

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:

```lisp
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 < p < n:

```lisp
(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]:

```lisp
DEFINEQ((FACTORIAL
  (LAMBDA (N) (IFPN=0 THENN 1 ESLE N*(FACTORIAL N-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.

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 to the Lisp community 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-10s, 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, whereas 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 ten 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 with not only 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.

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:

<table>
<thead>
<tr>
<th>Interlisp</th>
<th>MacLisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMBDA</td>
<td>spread</td>
</tr>
<tr>
<td>LAMBDA</td>
<td>nospread</td>
</tr>
<tr>
<td>NLAMBDA</td>
<td>spread</td>
</tr>
<tr>
<td>NLAMBDA</td>
<td>nospread</td>
</tr>
</tbody>
</table>

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
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.

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:

```lisp
DEFINISH ((FACTOIRIAL
            (LAMBDA (N)
               (COND [([ZEPO N) 1]
                    (T (TIMES N (FACTOIRIAL (SUB1 N)

MacLisp and Interlisp came into existence about the same time and lasted about as long as each other. They differed in their user 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 were educated by MIT, CMU, and Stanford. The principal implementations ran on the same machines; one had cachet as the Lisp with the friendly environment and 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.

6.2.4 The Early 1970s

Though MacLisp and Interlisp dominated the 1970s, 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 [Utah 1982]. Standard Lisp was defined 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.

Later, Hearn and his colleagues discovered that for good performance they needed more control over the environment and the compiler, so Portable Standard Lisp (PSL) was born. Standard Lisp attempted to piggyback on existing Lisps, whereas 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 1970s, PSL ran—and ran well—on more than a dozen different types of computers.

PSL was implemented using two techniques. First, it used a system implementation language called SYSLISP 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 1970s and on into the mid-1980s, 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 Hewlett-Packard in the mid-1980s.

At Stanford in the 1960s, an early version of MacLisp was adapted for their 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. Lisp 1.6 disappeared during the mid-1970s, one of the last remnants of the Lisp 1.5 era.

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 during the early to mid-1970s by some folks at Stanford, as well as at other institutions.

In 1976 the MIT version of MacLisp was ported to the WAITS operating system by Gabriel at the Stanford AI Laboratory (SAIL), which was directed at that time by John McCarthy.

6.2.5 The Demise of the PDP-10

By the middle of the 1970s 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-10s and DEC-20s) 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 6.2.6). The other response was to use commercial computers (stock hardware) 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; 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 primary VAX Lisp dialects developed in the late 1970s were VAX Interlisp, PSL (port 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.
NIL [Burke 1983], intended to be the successor to MacLisp, was designed by Jon L White, Steele, 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 because of the central role already played in the Lisp language by the atomic symbol named NIL. 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 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]. Close cooperation on this project was aided by the fact that Steele rented a room in Gabriel’s home. 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].

In France in the mid-1970s, Patrick Greussay developed an interpreter-based Lisp called Vlisp [Greussay 1977]. 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. Vlisp’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. (Vlisp was probably the first production-quality Lisp to support general tail recursion removal. Other dialects of the time, including MacLisp, did not support tail recursion removal in general situations only, in a manner not guaranteed predictable.)

Vlisp was the precursor to Le_Lisp, one of the important Lisp dialects in France and Europe during the 1980s; though the dialects were different, they shared some implementation techniques.

At the end of the 1970s, 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 (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.

6.2.6 Lisp Machines

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 the duality of load and store access based on functions, and 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 a1 ... an) is called in load
mode, it should return a value; if called in store mode, as in \((f \ a_1 \ldots \ a_n \ v)\), 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 \texttt{RPLACA}:

\begin{verbatim}
(LAMBDA (X Y) (SETFQ (CAR X) Y))
\end{verbatim}

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


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 nonprohibitive cost (less than \$70,000 per machine), single-user operation, a 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 an improved programming environment exploiting large, bit-mapped displays.

The CONS machine was built; then a subsequent improved version named the CADR (an in-joke—\texttt{CADR} means "the second one" in Lisp) was designed and some dozens of them were built. 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 Machine, Inc. (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 five years.

Although 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 (heap allocation) 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. (Although there was a great deal of theoretical work on interleaved and concurrent garbage collection during the 1970s [Steele 1975; Gries 1977; Baker 1978; Cohen 1981], 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 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. Although 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 (see Section 6.3.3)

- Backquote, a syntax for constructing data structures by filling in a template

- 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, nonhierarchical 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 1981]).

- SETF, a facility for generalized variables

The use of SETF throughout Common Lisp—a later dialect of Lisp and the most popular—can be traced through Symbols 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 that 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 \((f\ldots)\), then the form of the write is \((\text{setf} \ (f\ldots) \ \text{newvalue})\), and that is all the programmer needs to know about reading and writing data.

Later, in the Common Lisp Object System (CLOS), this idea was extended to methods. If a method \(M\) is specified to act as a reader and is invoked as \((M\ \text{object})\), then it is possible to define a writer method that is invoked as \((\text{setf} \ (M\ \text{object}) \ \text{newvalue})\).

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 involvement with Smalltalk, an early object-oriented language. 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.
6.2.6.2 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 Xerox 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 [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 undertook 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.

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 1970s. 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 1980s, 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 1970s BBN also built a machine, the Jericho, that was used as an Interlisp engine. It remained internal to BBN.

6.2.6.3 Comments on Early Lisp Machine History

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 quality 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 1980s, 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.

6.2.7 IBM Lisps: Lisp360 and Lisp370

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

In the early 1960s, 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-6 (and its successor, 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 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 a PDP-10.)

Moreover, in the early 1960s, 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 Cyril Alberga, Martin Mikelsons, Richard W. Ryniker II, and Mark Wegman; they served primarily themselves and a few research groups, such as the SCRATCHPAD symbolic computation 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. 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 IBM 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. Later, Mark Wegman and his colleagues at Yorktown extended this environment to include screen-based—rather than line-based—interaction and editing.

Improvements were made to the underlying Lisp system, such as good performance and the separation of the compilation and run-time 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 and featured 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 1980s in favor of Common Lisp.

6.2.8 Scheme: 1975–1980

The dialect of Lisp known as Scheme was originally an attempt by Gerald Jay Sussman and Steele during the Autumn of 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 wrote a tiny Lisp interpreter and then added 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. (Some of the issues and necessary mechanisms had already been explored by Joel Moses [Moses 1970].) It appeared that such a mechanism would be needed anyway for keeping track of acquaintances for actors. Lexical scoping 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 beginning 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

\[
(\text{define factorial}
\begin{align*}
(\lambda (n) \ (\text{if} \ (\neq n \ 0) \\
\ 1 \\
\ (* n \ (\text{factorial} \ (- n \ 1)))))
\end{align*}
\]

or the equivalent actor

\[
(\text{define actorial}
\begin{align*}
(\alpha (n \ c) \ (\text{if} \ (\neq n \ 0) \\
\ (c \ 1) \\
\ (\text{actorial} \ (- n \ 1) \ (\alpha (f) \ (c \ (* f n))))))
\end{align*}
\]

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 six-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 1976a] 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 1975]. 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 1976b] 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 6.4), 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 [Wand 1977] and they exchanged papers with Sussman and Steele during this period.

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, 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. 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, as these topics were soon treated adequately by other researchers. Although 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 [Church 1941]) 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, because 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.)

6.2.9 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 Machine, 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, 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 laboratories mostly located on the West Coast. BBN was constructing its Interlisp machine, 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 S-1 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).

6.2.10 Early Common Lisp

If there had been 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.

6.2.10.1 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 picture 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 powwow 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. Second, Gabriel and White approached Steele, then at CMU and
affiliated with the SPICE Lisp project. The three of them were all associated one way or another with
the S-1 NIL project. A few months later, Gabriel, Steele, White, Fahlman, William Scherlis (a
colleague of Gabriel’s then at CMU), and Rodney Brooks (part of the S-1 Lisp project) met at CMU,
and some of the technical details of the new Lisp were discussed. The new dialect was to have the
following basic features:

- Lexical scoping, including full closures
- Multiple values, like those in Lisp-Machine Lisp, but perhaps with some modifications for
  single-value-forcing situations
- Separate value and function cells (a Lisp-2) [Gabriel 1988] (See Section 6.2.12.4.)
- `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 those of Lisp-Machine Lisp
- No dynamic closures (closures over “special” variables, which are dynamically bound; also
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 nonobvious 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 the search 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.” (Such names as “Vulgar Lisp” were then bandied about.)

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, among 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.

The PSL group did not fully join the Common Lisp group and the Franz group did not join at all.
The Lisp370 group was, through oversight, not invited. The Interlisp community sent an observer.
ARPA was successfully pulled into supporting the effort.
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.

A 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 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 of 1981, 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 during that summer.

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 key decisions were made. This was followed by a face-to-face meeting in November of 1981, where the final decisions on the more difficult questions were settled.

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

The e-mail discussions were often in the form of proposals, discussions, and counterproposals. Code examples from existing software or proposed new syntax were often exchanged.

All e-mail was archived on-line, so everything was available for 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, for example, 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 had gone too far or had little support.
This meant that some 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 [Raymond 1991]. 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, such as 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 6.2.12.) This is the issue of modularization, which had two aspects: whether Common Lisp should be divided into a core language plus modules, and 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.

Common Lisp was not broken into a core language plus layers, and the white/yellow/red pages division never materialized.

Four more drafts were made—the Colander Edition (July 29, 1982), the Laser Edition (November 16, 1982), the Excelsior Edition (July 15, 1983), and the Mary Poppins Edition (November 29, 1983). Three of 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

As for the Excelsior Edition, recall that "excelsior" is not only a term of exhortation but also a name for shredded wood or paper suitable for use as packing material.

Virtually all technical 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.

The declared goals of the Common Lisp Group, paraphrased from CLTL1 [1984], were the following:

- 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. Although 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.
- 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.

6.2.10.2 Early Rumblings

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 language 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, as 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 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 compilers 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 CLTL1: Fahlman, Gabriel, Moon, Steele, and
Weinreb. 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.”

6.2.10.3 The Critique of Common Lisp

At the 1984 ACM 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 Brooks and Gabriel 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; many of those critiques
quoted this first one. The high points of the paper reveal a series of problems that would plague
Common Lisp throughout the decade.
This theme reappeared in the history of Common Lisp through 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 have officially or unofficially declared themselves “unCommon Lisps” at one time or another are Lisp370, Scheme, EuLisp, and muLisp. This was 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 are that it is a Lisp-1 (no separate function and variable namespaces), has a CLOS-style 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. 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.

6.2.11 Other Lisp 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.

6.2.11.1 Lisp Dialects on Stock Hardware

Portable Standard Lisp spread to the VAX, DECsystem-20, a variety of MC68000 machines, and the Cray-1. Its Emode environment (later Nmode) proved appealing to Hewlett-Packard, which “produced” 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.

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 busy 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 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 U.S. for Lisp. Lisp machines were dominant for Lisp in the U.S., but in Europe these machines were not as available and
often were prohibitively expensive. Research labs in Europe frequently acquired or were given a range of peculiar machines (from the U.S. perspective). Therefore, portability was a must.

The experience with Vi1isp 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 that of Franz Lisp, the most comparable alternative. On a VAX-11/780, Le_Lisp performed about as well as Zetalisp on a Symbolics 3600.

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.

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, Indiana University, and Rice University. Most of these groups were started by MIT or Indiana graduates who joined the faculty at these schools.

At MIT, under the guidance of Hal Abelson and Gerry Sussman, Scheme was adopted for teaching undergraduate computing. The book *Structure and Interpretation of Computer Programs* [Abelson 1985] became a classic and vaulted Scheme to notoriety in a larger community.

Several companies sprang up that made commercial implementations of Scheme. Cadence Research Systems was started by R. Kent Dybvig; its Chez Scheme ran on various workstations. At Semantic Microsystems, Will Clinger, Anne Hartheimer, and John Ulrich produced MacScheme for the Apple Macintosh. PC Scheme, from Texas Instruments, ran on the IBM PC 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 "Scheme 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 such as call-with-current-continuation were added, the rate of change of Scheme slowed down because of 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 emerged a series of Revised Reports, called The Revised, Revised, Revised, Revised Report on Scheme. In late 1991, The Revised, Revised, Revised, Revised, Revised Report on Scheme was written and approved; it is affectionately called "R4RS."

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

6.2.11.2 Zetalisp

Zetalisp was the name of the Symbolics version of Lisp-Machine Lisp. Because the 3600—the Symbolics second-generation Lisp machine, which is described in the following—was programmed almost entirely in Lisp, Zetalisp came to require a significant set of capabilities not seen in any single Lisp before 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 nonhierarchical object-oriented language, 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 continued to play a key role.

The features of Flavors were 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 predetermined 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. It may be loosely characterized as FORTRAN FORMAT statements gone berserk. SETF is discussed in Section 6.2.6.)

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.

6.2.11.3 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 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 proprietary 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 (and later a 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 was most successful (as measured by number of installations at the end of the pre-Common-Lisp era), followed by Xerox and TI, though TI possibly could have claimed the most installed machines on the basis of one or two large company purchases.

Symbolics is the 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. 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 1980s workstations were an oddity, and they were generally so computationally underpowered that many people 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, because 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, notably hackers Howard Cannon, Tom Knight, David Moon, and Dan Weinreb.

One of the factors in the adoption of Symbolics Lisp machines and Zetalisp was the fact that the first 3600s 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 [Baker 1978], but because the address space was so large, ordinary programs did not exhaust memory for several days and intensive ones could run for about eight 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 3600s. 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 [Ungar 1984]. 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 it 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 was already established 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's 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.

The early history of the Xerox D-machines is discussed in the preceding. 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 (no relation to their current implementation of the SPARC microprocessor architecture!), which ran Spice Lisp on a Motorola MC68020 microprocessor. Later, they decided to go the pure Lisp machine route and introduced the Explorer, 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 some capital into LMI, prolonging its existence. The Explorer had good price-performance and decent absolute performance. 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.

6.2.11.4 MacLisp on the Decline

Though MacLisp was still in widespread use and spreading a bit, development halted in the early 1980s. At this point, MacLisp ran on ITS, Multics, Tenex, TOPS-10, TOPS-20, and WAITS. (All but Multics were various operating systems for the PDP-10.)

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, although MACSYMA was an interesting application of Lisp, MacLisp itself 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 as MIT saw fit to provide.

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 S-1 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 [Gabriel 1984a]. This environment defined a protocol that allowed MacLisp and E, the Stanford display editor that 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 time sharing 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 "editor 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 remained profitable until its dissolution in the late 1980s or early 1990s.

With NIL funding stoppage, Jon L White joined Xerox to work on Interlisp. A stranger situation is difficult to imagine. First, White was so clearly an Easterner that the California lifestyle would seem too foreign for him to accept. Second, the intense rivalry between MacLisp and InterLisp over the years would seem to prevent their working together.


The period just after the release of *Common Lisp: The Language* [CLTL1 1984] 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 from that of the Lisp machines.
- The venture capital community, which, looking at the success of companies like Sun, was 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.
- Computer scientists were turning into entrepreneurs in droves, spurred by the near-instant success of their colleagues in such companies as 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 about 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. The allure of AI and Lisp attracted both businessmen and venture capitalists.

As early as 1984—the year CLTL1 was published—several companies were founded to commercialize Common Lisp. These included Franz Inc.; Gold Hill Computers, Inc.; 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, including 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 had a compiler that compiled to C, which was compiled by the C compiler. This Lisp was licensed essentially free, and the Common Lisp companies suddenly had a surprising competitor. (Surprising, because it appeared just as CLTL1 came out—the implementation was based on the Mary Poppins draft. KCL was also notable because it was implemented by outsiders, Taiichi Yuasa and
Masami Hagiya, solely on the basis of the specification. This effort exposed quite a number of holes and mistakes in the specification that had gone unnoticed by those who, having participated in the historical development of Common Lisp, consciously or unconsciously corrected for such mistakes as they went along on the basis of additional shared knowledge.)

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.

6.2.12.1 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 Vaxima. Franz Lisp was the most popular dialect of Lisp on the VAX until plausible Common Lisps appeared. Franz decided 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 was a Ph.D. student of Fateman's at the University of California at Berkeley in the mathematics department; Foderaro, having already obtained his Ph.D. under Fateman, became the primary architect and implementor of the various Lisps offered by Franz, Inc. Fateman, one of the original implementors of MACSYMA at MIT, carried the MacLisp/Lisp torch to Berkeley; he was responsible for the porting of MACSYMA to the VAX. 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. Barber had spent the year before founding Apiary/Gold Hill 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, it was close enough to Common Lisp that he and Hewitt 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; worse yet, as the Lisp was transformed into a Common Lisp, its quality apparently dropped. At the same time, the so-called "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. Gold Hill sold direct as well.

"AI winter" is the term first used in 1988 to describe the unfortunate commercial fate of AI. From the late 1970s until the mid-1980s, 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.

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 from that of the other Common Lisp companies. Instead of starting with the Spice Lisp source code, Lucid wrote an implementation of Common Lisp from scratch; moreover, it adopted an OEM strategy. (The OEM idea is to make arrangements with a 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 collects royalties.) Lucid quickly struck OEM deals with Sun, Apollo, and Prime. This was possible because 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.

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, and maintenance fees as well as royalties. Getting Sun as an OEM was the key to Lucid's survival, because Sun workstations developed the cachet that was needed to attract customers to the Lisp. Sun was always regarded as leading-edge, 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 the AI winter hit, Lucid began diversifying into other languages (C and C++) and programming environments.

6.2.12.2 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 1980s, 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 1980s, HP realized that PSL was not the winner and they needed to provide a Common Lisp. They chose Lucid to provide it and reduced their own engineering staff, choosing to focus more on marketing. Because 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 1990s, in the midst of AI winter, DEC also decided to abandon its own efforts and also chose Lucid.

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. Because of Fahlman's relationship with Lucid (a founder), a contract was eventually written for Lucid to port its Lisp to the RT for IBM.

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.

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

Envos put out a real Common Lisp implementation and sold the InterLisp-D environment. But Envos 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 the same software Envos had been, but without direct funding. The funding was provided by servicing Xerox's Lisp customer base (maintenance).
6.2.12.3 DARPA and the SAIL Mailing Lists

Right after CLTL1 was published in 1984, ARPA (renamed DARPA) took a real interest in Common Lisp. They sponsored a community meeting and encouraged further development of Common Lisp into a full development system, including an object-oriented extension, multitasking, window systems, graphics, foreign function interfaces, and iteration. It seemed that DARPA wished for a resurgence of Lisp and was willing to provide some funds to help fulfill its wish.

After the meeting, new mailing lists were set up at SAIL for discussion of these topics and, though most of the lists were quiet, some witnessed interesting discussions.

6.2.12.4 The Start of ANSI Technical Committee 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: Common Lisp must be standardized, DARPA announced, and Robert Mathis, the convenor of the ISO working group on the Ada programming language, was to head up this effort.

The reason for this sudden need was that the European Lisp community was planning to launch their own Lisp standardization effort at ISO to head off the spread of Common Lisp. Mathis, with his storehouse of international experience, seemed to DARPA the natural choice to head up the response, and the stunned, confused, and soon-to-be-previously-pastoral Common Lisp group could think of little else to do but go along.

The period from Spring of 1986 until Spring of 1992 was a combination of political wrangling and interesting Lisp development. As usual, the impetus behind the Lisp development was to increase the expressive power of Lisp. The political wrangling centered around two different objectives: within Common Lisp, each individual strived to put his or her mark on the language; outside Common Lisp, various groups tried to minimize the size of Lisp to guarantee its survival—both academic and commercial.

A few months after the December 1985 meeting there was a meeting at CBEMA (the Computer and Business Equipment Manufacturers Association) in Washington, D. C. (CBEMA serves as the secretariat for X3, an Accredited Standard Committee for Information Processing Systems operating under the procedures of ANSI, the American National Standards Institute. Among the better-known technical committees under X3 are X3H3 (computer graphics, including PHIGS), X3J3 (programming language FORTRAN), X3J4 (programming language COBOL), and X3J11 (programming language C).) The goals of standardization were discussed, and the most important topic, which was pushed by DARPA, was whether to merge with the Scheme activities—the technical issues surrounded the treatment of macros and whether there was a separate namespace for functions separate from ordinary variables. The goals of the new group, soon to become Technical Committee X3J13 for Programming Language Common Lisp, were also discussed.

The point about namespaces is important to understanding the debate between Lisp dialect proponents. A namespace is a mapping between an identifier (string of characters) and its meaning. In Common Lisp there are a number of namespaces—variables, functions, types, tags, blocks, and catch tags, among others. If a Lisp has separate namespaces for variables and functions, users are allowed to use variable names that also name functions, because the evaluation rules specify the namespace in which to look for the meaning. In a Lisp with a single namespace, the user must be careful when creating variable names to avoid shadowing a function name. This issue is important for macros, which in effect must carefully decide what a free variable is intended to mean. Although there are many namespaces in Lisp, because the variable and function namespaces are where the problems lie in practice, this issue reduces to the question of whether variable names and function names belong
GUY L. STEELE JR. & RICHARD P. GABRIEL

to one namespace or two separate namespaces. It is therefore often referred to as the "Lisp-1 versus Lisp-2 debate." A Lisp-1 is a Lisp where functions and variables are in the same namespace, and a Lisp-2 is a Lisp where they are in separate namespaces.

The effort to merge the Scheme and Common Lisp communities was launched on two fronts. One was to try to come up with a solution to the macro problem that a Lisp-1 causes. This problem is that with only one namespace it is relatively easier to stumble across an unintended name conflict leading to incorrect code. The key Common Lisp leaders felt that if the macro problem could be solved, Common Lisp could survive a transition from Lisp-2 to Lisp-1. The other front was to try to convince the Scheme community that this was a good idea.

On the first front, Gabriel and Kent Pitman produced a report detailing the technical issues involved in macros [Gabriel 1988]. Several technical solutions appeared around the same time, the most promising being described in Kohlbecker's dissertation [Kohlbecker 1986b].

On the second front, Gabriel and Will Clinger approached the Scheme community, which soundly rejected any association with the Common Lisp community. (Sadly, though there were several other attempts to bring the two communities together, there has never been any serious dialogue between the two groups.)

By the end of 1986, it was clear that the European Lisp community would eventually produce a new Lisp dialect, which would be informally called EuLisp, and that the same community intended to start an ISO effort to standardize this dialect.

EuLisp is a dialect of Lisp defined in layers, with a very small kernel language and increasingly larger ones, the goal being to have a Common-Lisp-sized layer. EuLisp has an object-oriented facility, modules, multitasking, and a condition system. It is a Lisp-1. (A condition system is a facility for defining and handling user exceptions and for handling system exceptions. In Common Lisp, this facility provides a mechanism for executing user-defined code in the dynamic context of the error.)

The most important technical development during this period was the Common Lisp Object System (CLOS). In 1986 four groups began to 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 (HP) [Kempf 1987]. After a six-month battle, a group was formed to write the standard for CLOS based on CommonLoops and New Flavors. This group was David A. Moon (Symbolics), Daniel G. Bobrow (Xerox), Gregor Kiczales (Xerox), Sonya Keene (Symbolics, a writer), Linda DeMichiel (Lucid), and Gabriel (Lucid). Also, certain others contributed informally to the group: Patrick Dussud (TI), Jim Kempf (HP), and Jon L. White (Lucid). The CLOS specification took two years, and the specification was adopted in June of 1988 with no changes.

CLOS has the following features:

- **Multiple inheritance using a linearization algorithm to resolve conflicts and to order methods.** Multiple inheritance provides a mechanism to build new classes by combining mixins, which are classes that provide some structure and behavior. Programming with multiple inheritance enables the designer to combine desired behaviors without having either to select the closest existing class and modify it or to start a fresh single inheritance chain.

- **Generic functions whose methods are selected based on the classes of all required arguments.** This is in contrast to the message-passing model in which a message is sent to a single object whose class selects the method to invoke.

- **Method combination, which provides the mechanism to take behaviors from component parts and blend them together.** Method combination is an important aspect of multiple inheritance,
because each combined class can provide part of the behavior needed, and the programmer need not code up a combining method to use existing methods.

- Metaclasses, whose instances are classes and which are used to control the representation of instances of classes.
- Meta-objects, which control the behavior of CLOS itself. CLOS can be viewed as a program written in CLOS. Because any CLOS program can be customized, so can CLOS itself.
- An elaborate object creation and initialization protocol that can be used to provide user customization of the instance creation, change class, reinitialization, and class redefinition processes.

During the deliberations of X3J13, a number of other additions were made to Common Lisp: an iteration facility, a condition system, a better specification of compilation and evaluation semantics, and several hundreds of small cleanups.

The X3J13 process was unlike the Scheme process. The Scheme process allowed any person to veto an addition. The X3J13 process went by majority vote. This allowed a great deal of log-rolling, and some committee members were eager to put their mark on Common Lisp.

The iteration facility, called LOOP, consists of a single macro that has an elaborate pseudo-English or COBOL-like syntax. The debate on this facility was at times intense, especially when Scott Fahlman was still active in Common Lisp. Because of its non-Lispy syntax, it was (and remains) easy to ridicule. (See Sections 6.3.2 and 6.3.5.1.)

The condition system further developed the exception handling capabilities of Common Lisp by introducing first-class conditions and mechanisms for defining how conditions of certain classes are to be handled—either automatically or with human intervention. Adoption of this facility was made easier by the adoption of CLOS, which paved the way for a cleaner formulation of the basic mechanisms. Nevertheless, the condition system was not completely CLOSified and cleaned up. For example, clauses that appear syntactically to be method definitions and hence should be selected based on class specificity are actually treated like COND clauses.

In 1987 ISO created a working group called WG 16 to begin the process of standardizing Lisp at the international level. The two primary contenders were EuLisp and Common Lisp.

The political goal of EuLisp was to displace Common Lisp from Europe. Because U.S. standards had such a strong influence in Europe and because the only standards organization with real clout in Europe was ISO, this route was dictated.

The intellectual goal was a clean, commercial-quality, layered Lisp dialect for the future. EuLisp appears to have met its goals and many consider it one of the nicer Lisp definitions, though there are still no commercial implementations of it.

For five years, the U.S. managed to keep any progress from being made in the ISO committee until, in 1992, a compromise was worked out in which, essentially, a near subset of Common Lisp and of CLOS would form the basis of a kernel Lisp dialect.

In 1988, DARPA called another Lisp meeting to discuss bringing the Scheme and Common Lisp communities together but, as with earlier attempts, this failed primarily because the Scheme community did not want to have anything to do with Common Lisp. Attending this meeting were Daniel G. Bobrow, Scott Fahlman, Gabriel, Bill Scherlis, Steve Squires, and Gerry Sussman.

In 1989, Scheme began an IEEE standardization process, which culminated in 1991 with both an IEEE and ANSI standard [IEEE 1991], the latter after a virtually unannounced public review period. The structure of the Scheme standards is that the official standard lags the informal R^n Report, so that the standard corresponds to the R^{n-1} Report when the R^n Report is current.
Also in 1989, the first non-intrusive garbage collectors appeared from the companies Lucid and Franz. The Lucid collector is an ephemeral garbage collector based on a combination of ideas from Smalltalk-generation scavengers and the Symbolics ephemeral garbage collector [Sobalvarro 1988]. The appearance of these collectors seemed to have the effect of increasing the legitimacy of stock-hardware Lisp companies to the same or higher level than the Lisp machine companies. This was because the Lisp machine companies encouraged the belief that stock-hardware Lisps could never have the performance—particularly for garbage collection—that the special-purpose-computer Lisps could have. When this was proven wrong, the Lisp machine companies suffered.

In 1991, the R4 Report included a specification of hygienic macros, the first partially standardized macro facility in Scheme.

As X3J13 progressed and the power of general purpose workstations increased—largely due to development of fast RISC processors—stock hardware Lisp companies dominated, forcing most of the Lisp machine companies either out of business or out of their leadership position. Also, the deterioration of the Lisp market and the general decline of the economy in the U.S. combined to enable the smaller software-based Lisp companies to survive—customers were not willing to buy expensive “dedicated” computers and then spend money maintaining them.

In April of 1992 X3J13 delivered to X3 SPARC (the authorizing body for X3J13—no relation to the microprocessor architecture of the same name) its draft for Common Lisp. At the same time, ISO WG 16 produced the first draft of its kernel Lisp.

The ANSI draft for Common Lisp is immensely large—well over 1000 pages. We are told that one official at X3 SPARC, on first seeing it, gasped: “It’s bigger than COBOL! Much bigger!”

Though this might seem funny, it shows how increasing desire for expressiveness, intensified attention to getting the details right (even for details that almost never matter), the need for individuals to make their mark, and a seemingly deliberate blind eye towards commercial realities can lead to an unintended result—a large, unwieldy language that few can completely understand.

### 6.3 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.

But for the constraints of space, we would also have addressed many other topics that have figured prominently in the technical development of Lisp or have been addressed in unusual ways in the context of the Lisp language:

- continuations
- reification and reflection
- garbage collection
- stack management
- record structures
- oblist, obarray, packages, modules
- hash tables vs. property lists
- structure editors
- pretty-printing
- program tracing and debugging
- closures (lexical and dynamic)
- nonlocal exits and UNWIND-PROTECT
- parallel processing
- object-oriented programming

Each of these topics has a story that spans decades and interacts with the development of language theory and other programming languages. Here we must content ourselves with a few topics that are representative of the concerns of the Lisp community.
6.3.1 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 contain 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 did not need to 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 Steele, who had to find every place in the assembly-language kernel of MacLisp where this mattered.

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, p. 2.2]:

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, [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 1964]. 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 the construction

\[(\text{if (car } x\text{)} (+ \text{(car } x\text{)} 1))\]

as a bad pun, preferring the more explicit

\[(\text{if (not (null (car } x\text{))) (+ (car } x\text{)} 1))\]

The Revised Revised Report on Scheme [Clinger 1985b] defined three distinct quantities \texttt{nil} (just another symbol); (), the empty list; and \texttt{#false}, the Boolean false value (along with \texttt{#true}, the Boolean true value). However, in an interesting compromise, all places in the language that tested for true/false values regarded both () and \texttt{#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 \texttt{nil}.

The Revised\textsuperscript{3} Report on the Algorithmic Language Scheme [Rees 1986] shortened \texttt{#false} and \texttt{#true} to \texttt{#f} and \texttt{#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 \texttt{nil}.

The recently approved IEEE standard for Scheme specifies that \texttt{#f} and \texttt{#t} are the standard false and true values, and that all values except \texttt{#f} count as true, "including \texttt{#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 \texttt{NIL}. Nevertheless, the question continues to be debated.

The question of \texttt{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 \texttt{NIL}. (A running joke was that NIL (New Implementation of Lisp) unburdened \texttt{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 [Raymond 1991] carried the day.

It is worth noting that Lisp implementors have not been tempted to identify \texttt{NIL} with the number 0 (as opposed to the internal \texttt{address} 0), with one notable exception, a Lisp system for the PDP-11 written in the 1970s by Richard M. Stallman, in which the number 0 rather than the symbol \texttt{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 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 a great deal of discussion 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.

6.3.2 Iteration

Although Lisp, according to its PR, 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. In some cases this was driven by the desire to emulate styles of programming found in ALGOL-like languages [Abrahams 1966] 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 1980s, so performance was an issue.

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

\[
\text{(do \ var \ init \ step \ test \ . \ body)}
\]

means the same as

\[
\text{(let ((var init))}
\text{(block (when test (return))}
\text{ (progn \ . \ body)}
\text{ (setq \ var \ step)))}
\]

Thus the FORTRAN DO loop

\[
\text{DO 10 \ J=1,100}
\text{ IF (A(J) \ .GT. 0) SUM = SUM + A(J)}
\text{ 10 CONTINUE}
\]

could be expressed as

\[
\text{(DO J \rightarrow 0 TO 100 \ (A J) \ WHEN (PLUSP (A J)))}
\text{ (SETQ TOTAL (+ TOTAL (AREF A J))))}
\]

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

\[
\text{(DO J \rightarrow 0 TO 99 \ (A J) \ WHEN (PLUSP (A J)))}
\text{ (SETQ TOTAL (+ TOTAL (AREF A J))))}
\]

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:

\[
\text{(FOR J \rightarrow 0 TO 99 \ SUM (A J) \ WHEN (PLUSP (A J)))}
\]

This returns the sum as its value rather than accumulating it into the variable TOTAL by side effect. (See Section 6.3.5.1 for further discussion of ALGOL-style syntax.)

Macros or other code-transformation facilities such as CLISP make this kind of extension particularly easy. Although 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 \texttt{FOR} or \texttt{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 \texttt{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 Steele. The result was a compromise. The first definition of Common Lisp [CLTL1 1984] included a \texttt{loop} macro with absolutely minimal functionality: it permitted no special keywords and was good only for expressing endless repetition of a sequence of subforms. It was understood to be a placeholder, reserving the name \texttt{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 \texttt{loop} [CLTL2 1990] 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) [CLTL2 1990; Waters 1984, 1989a, 1989b]. The example FORTRAN DO loop shown would be rendered using series as

\begin{verbatim}
(collections (choose-if #'plusp
  (#M(lambda (j) (a j))
    (scan-range :start 0 :below i00) ) )

The call to \texttt{scan-range} generates a series of integers from 0 (inclusive) to 100 (exclusive). The notation \texttt{#M} means "map"—the following function is applied to every element of a series, producing a new series. The function \texttt{choose-if} is a \texttt{filter}, and \texttt{collect-sum} returns the sum of all the elements in a series. Thus series are used in a functional style, reminiscent of APL:

\begin{verbatim}
+/((T>0)/T-A[100])
\end{verbatim}

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

\begin{verbatim}
(generator (scan-range :start 0 :below 100))
\end{verbatim}

produces an object that delivers successive integers 0 through 99 when repeatedly given to the extraction function \texttt{next-in}. The complete example might be rendered as

\begin{verbatim}
(let ((num (generator (scan-range :start 0 :below 100))))
  (gathering ((result collect-sum)
    (loop (let ((j (next-in num (return result)))
      (if (plusp j) (next-out result j)))))))
\end{verbatim}

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. Generators and gatherers emphasize use of control structure rather than functional relationships.
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 of 1972:

```
(do ((varl initl 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, give the variables of the values of the step forms, and repeat.

The beauty of this construct is that it allows the initialization and stepping of multiple variables without use of 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)))
      (if (= 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.

Although 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 (varl var2 ... varn)
               (cond (test . result)
                     (t (progn . body)
                        (the-loop step1 step2 ... stepn))))))
     (the-loop initl 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.
6.3.3 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 (with permission):

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.
3. define will be modified to make macro expansions.

Examples:

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

\[
\text{MACRO} \((\text{CSETQ} \ (\text{LAMBDA} \ (\text{FORM}) \ (\text{LIST} \ (\text{QUOTE} \ \text{CSET}) \ (\text{LIST} \ (\text{QUOTE} \ \text{QUOTE}) \ (\text{CADR} \ \text{FORM}))) \\
\text{(CADDR} \ \text{FORM})))\)\]

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

\[
x := \text{cons}[\text{form};x]
\]

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

\[
\text{(STASH} \ \text{FORM} \ x) \text{Stash may be defined by:}
\]

\[
\text{MACRO} \ ((\text{STASH} \ (\text{LAMBDA} \ (\text{FORM}) \ (\text{LIST} \ (\text{QUOTE} \ \text{SETQ}) \ (\text{CADAR} \ \text{FORM}) \ (\text{LIST} \ (\text{CONS} \ (\text{CADR} \ \text{FORM}) \ (\text{CADAR} \ \text{FORM})))))))
\]

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.

\[
\text{MACRO} \ \text{enter}[\text{form}] = \text{list}[\text{STASH};\text{list}[\text{CONS};\text{cadr}[\text{form}];
\text{caddr}[\text{form}]];\text{cadddr}[\text{form}]])
\]

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:

\[
\text{macro}[1] = \text{deflist}[1;\text{MACRO}]
\]

The new define is a little harder:

\[
\text{define}[1] = \text{deflist}[\text{mdef}[1];\text{EXPR}]
\text{mdef}[1] = \{
\text{atom}[1] \rightarrow 1,
\]

\[
274 \text{ CHAPTER VI}
\]
4. The macro for select illustrates the use of macros as a means of allowing functions of an arbitrary number of arguments:

\[
\text{MACRO}
\]

\[
\text{select} \{\text{form}\} = \lambda[[g];
\text{list}[[\text{LAMBDA};\text{list}[g];\text{cons}[\text{COND};
\text{maplist}[\text{cddr}[\text{form}];\lambda[[1];
\text{null}[\text{cdr}[1]] \rightarrow \text{list}[\text{T};\text{car}[1]];
\text{T} \rightarrow \text{list}[[\text{EQ};g;\text{caar}[1]];\text{cadar}[1])]]]]);\text{cadr}[\text{form}]]][\text{gensym}[]]
\]

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 1960s. 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 1964b]. Inspection of the MDEF function in the compiler code [Saunders 1964a, p. 311] reveals that the error in processing PROG statements had been repaired: mdef[caddr[1]] was replaced by mdef[ccaddr[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[ccaddr[1]] and subsequently a wrong correction was made.) Unfortunately, the use of mdef[ccaddr[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.

Macros of this kind were an integral part of the design of Lisp 2. Abrahams et al. remark that by 1966 macros were an accepted part of Lisp 1.5 as well [Abrahams 1966].

A similar sort of computed macro appeared in the MIT PDP-6 Lisp, but macro 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; the pretty-print-
ing 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 \texttt{(si:displaced \ macro-call expansion)}. The macro \texttt{si:displaced} transparently returns its second argument form, so everything behaves as if the expansion itself had replaced the macro call. Although expanding the call to \texttt{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 \texttt{displace}). The Lisp Machine pretty-printer recognizes calls to \texttt{si:displace} and prints only the original macro call.

BBN-Lisp [Teitelman 1971] had three kinds of macro: open, computed, and substitution (described later). A macro definition was stored in the property list of its name under the \texttt{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 \texttt{ABS}:

\begin{verbatim}
(LAMBDA (X) (COND ( (GREATERP X 0) X) (T (MINUS X)) ))
\end{verbatim}

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 \texttt{LIST}:

\begin{verbatim}
(X (LIST (QUOTE CONS)
        (CAR X)
        (AND (CDR X)
              (CONS (QUOTE LIST) (CDR X))))
\end{verbatim}

The leading \texttt{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 6.2.3.)

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 \texttt{ABS} would look like this:

\begin{verbatim}
(((X) (COND ((GREATERP X 0) X) (T (MINUS X)))) )
\end{verbatim}

However, the call \texttt{(ABS (FOO Z))} would be expanded to

\begin{verbatim}
(COND ((GREATERP (FOO Z) 0) (FOO Z)) (T (MINUS (FOO Z))))
\end{verbatim}

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

\begin{verbatim}
(((LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) ) (FOO Z))
\end{verbatim}

which would evaluate \texttt{(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 1960s and 1970s. (See Section 6.4.) We return to the topic of template-based macros later.

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:

```lisp
<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):

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

```lisp
(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 1970s was to use a `LAMBDA` expression:

```lisp
(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.

To see how easily this can get out of hand, consider a macro for a simple `FOR` loop. A typical use would be this:
This should expand to
\[
\text{(do a 1 (+ a 1) (> a 100) (print a) (print (* a a)))}
\]

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 6.3.2.) In the MacLisp of the early 1970s one might define it as follows:
\[
\text{(defun for macro (x) (cons 'do (cons (cadr x) (cons (caddr x) (cons (llst (cons (cadddr x)) '+ (cadr x) 1) (list '> (cadr x) (cddddr x))) (cddddr x)))))}
\]

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 in a template of certain symbols 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:
\[
\text{(defmacro for (var lower upper . body) (cons 'do (cons var (cons lower (cons (list '+ var 1) (cons (list '> var upper) body))))))}
\]

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:
\[
\text{(defmacro for (var lower upper . body) '(do ,var ,lower (+ ,var 1) (> ,var ,upper) ,@body))}
\]
Note the use of \texttt{,@} to indicate splicing.

The backquote syntax was particularly powerful when nested. This occurred primarily within macro-defining macros; because these 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.

Backquote and \texttt{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 \texttt{FEXPR’s} in defining new language syntax [Pitman 1980].

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

\begin{verbatim}
(DEFUN INC MACRO (VAR . REST)
  " (SETQ ,VAR (+ ,VAR ,(IF REST (CAR REST) 1)))")
\end{verbatim}

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

\begin{verbatim}
(DEFUN INC MACRO (VAR &OPTIONAL (N 1))
  "(SETQ ,VAR (+ ,VAR ,N)))")
\end{verbatim}

This occurred in January of 1979, according to Lisp Archive, at which time MacLisp absorbed \texttt{DEFMACRO} and \texttt{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 \texttt{LET}:

\begin{verbatim}
(defun let (bindings . body)
  "((lambda , (mapcar #'car bindings) ,@body)
   ,@(mapcar #'cadr bindings)))")
\end{verbatim}

Note the use of \texttt{MAPCAR} for iterative processing of the bindings. This difficulty was not tackled by Lisp-Machine Lisp or Common Lisp; in that community \texttt{DEFMACRO} with \&-keywords is the state of the art today. Common Lisp did, however, generalize \texttt{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-1980s, 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 \texttt{\%\%foo\%\%} (though it is surprising how often great minds will think alike), by using \texttt{gensym} (as Hart did in his \texttt{select} example, shown in the preceding), or by using multiple obarrays or packages to avoid name clashes. However, none of these techniques provides an iron-clad guarantee. Steele pointed out that careful use of thunks could probably 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 1980s. 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-1980s 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. Although not as general as the separate matching and substitution facilities of \texttt{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, \texttt{INC} might be defined as

\begin{verbatim}
(extend-syntax (inc) ()
  ((inc x) (inc x 1))
  ((inc x n) (setq x (+ x n))))
\end{verbatim}

and \texttt{LET} as

\begin{verbatim}
(extend-syntax (let) ()
  ((let ((var value) ...) body ...) )
  ((lambda (var ...) body ...) value ...))
\end{verbatim}

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 \texttt{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.

\[
\text{(extend-syntax (loop while repeat) (exit)}
\]
\[
((\text{loop } el \ e2 ... \repeat)
\text{(call/cc (lambda (exit}}
\[
\text{((label foo (lambda () el \ e2 ... (foo))))))}
\]
\[
((\text{loop } el ... \text{ while } p \ e2 ... \repeat)
\text{(call/cc (lambda (exit}}
\[
\text{((label foo (lambda () el ... (unless p (exit #'f))}
\text{e2 ... (foo))))))))
\]

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 “el 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 [Church 1941]. Sometimes the rule of \( \alpha \)-conversion must be applied to rename
variables in a lambda-calculus expression so that a subsequent \( \beta \)-reduction will not produce a name
clash. One cannot do such renaming all at once; it is necessary to intersperse renaming with the
\( \beta \)-reductions, because a \( \beta \)-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 also 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? (We say this not only to illustrate the character of Lisp, but also to illustrate the character of Lispers. The Lisp community is motivated, in part, by an attitude of superiority to the competition, which might be another programming language or another dialect of Lisp.)

6.3.4 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 1964] had only fixnums; apparently the same is true of the M-460 Lisp [Hart 1964]. Lisp 1.5 on the 7090 had floating-point [McCarthy 1962], as did Q-32 Lisp [Saunders 1964b] and PDP-6 Lisp [PDP-6 Lisp 1967].)

We are still a little uncertain about the origin of bignums (a data type that uses a variable amount of storage 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 [CLTL2 1990] require them. Usually the algorithms detailed in Knuth Volume 2 are used [Knuth 1969, 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 HAILONG 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 1980s 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.)

Although 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 1977b, 1977c]; 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 [Fatemian 1973].
GUY L. STEELE JR. & RICHARD P. GABRIEL

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. This
was something of a sticking point with Scott Fahlman. A running joke was an acceptance test for
nascent Common Lisp implementations developed by 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

\[
(\text{factorial } 1000) \quad (\text{factorial } 999)
\]

If it responds 1000, it passes part 2. Third, you try \( \text{atanh} \, -2 \). 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 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 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—\text{truncate}, \text{round}, \text{ceiling}, and \text{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.
6.3.5 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.

6.3.5.1 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 [1981] noted:

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 1964]. 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:

```lisp
(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 ARPA-supported LISP 2 project aimed at providing Lisp with a syntax resembling that of ALGOL 60, citing the advantage that “ALGOL algorithms can be utilized with little change” [Abrahams 1966]. LISP 2 code for UNION in the style of our running example would look like this:

```lisp
SYMBOL FUNCTION UNION(X, Y); SYMBOL X, Y;
 IF NULL X THEN Y
 ELSE IF MEMBER(CAR X, Y) THEN UNION(CDR X, Y)
 ELSE CAR X UNION(CDR X, Y);
```

However, contemporary examples of LISP 2 code seem to emphasize the use of loops over recursion, so the following version might be more typical of the intended style:

```lisp
SYMBOL FUNCTION UNION(X, Y); SYMBOL X, Y;
 BEGIN
 SYMBOL Z ← Y;
 FOR A IN X DO
  IF NOT MEMBER(A, Y) THEN Z ← A . Z;
 RETURN Z;
 END;
```

(Of course, this version produces a result that is different when regarded as a list, although the same when regarded as a set.)

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 1970s [Wegbreit 1971, 1972, 1974]. The UNION function in EL1 looks like this:

```el1
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 \Rightarrow 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 1970s.

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:

```lisp
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)>I
              )))
```

In CLISP, - is a unary operator meaning NOT or NULL. X:1 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; I within such a list indicates splicing, so <A I B> means (CONS A B). Finally, the use of a final I to indicate the necessary number of closing parentheses (four, in this case), although 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.

```lisp
EXPR UNION (X, Y); %MLISP version of UNION
IF ~X THEN Y ELSE
  IF X[1] E Y THEN UNION(X[1], Y)
  ELSE X[1] CONS UNION(X[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 reimplementation for SCRATCHPAD, and 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):

```lisp
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
```

LISP SESSION 287
Here we have assumed the version of CGOL implemented at MIT, which stuck to the standard ASCII character set; the same definition using the Stanford extended character set would be:

```lisp
(define x "U", y, 14, 13;
  if ~x then y else if ~x ∈ y then βx ∪ y else βx ∪ y )
```

The "." represents CONS in both preceding examples; the delimiter ‹ (actually the ASCII “altmode” 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 preceding 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 [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 [1964]:

The one great cause of most of the incorrect results obtained in practice is an incorrect precedence being assigned to a function.

During the 1970s 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 6.4 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, so 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 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.)

6.3.5.2 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

```lisp
(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

```lisp
((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

```lisp
((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

```lisp
((lambda (quo rem) (print quo) rem) (/ 44 6))
```

returns only the remainder 2 after printing the quotient 7. 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.

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

```lisp
(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

```lisp
(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 (Len Bosack, Ralph Goren, David Posner, William Scherlis, Carolyn Talcott, Richard Weyhrauch, 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 1960s and early 1970s, rather than as a variant of Lisp; it enjoyed some popularity in Europe and was used to implement the logic programming language POPLOG [Mellish 1984].

### 6.3.5.3 Logic Programming and Unification

During the 1970s and on into the 1980s 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 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.

6.4 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 exercises:

1. Add a new arithmetic operator to the language similar to the one for Pythagorean addition in Knuth’s Metafont language [Knuth 1986]: \( a \div + b \) 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 \(+{}+{}+\); for APL, use one of the customary awful overstrikes such as \(+\).)

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):

```lisp
(defun ++ (x y) (sqrt (+ (* x x) (* y 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):

\[
\text{(defmacro switch (value &rest body) (let* ((newbody (mapcar #'(l~mhda (clause) '((gensym) ,@(rest clause))) body)) (switcher (mapcar #'(l~mhda (clause newclause) ' (, (first clause) (go , (first newclause))) body newbody)) ' (block switch (tagbody (case ,value ,@switcher) (break) ,(@(apply #'nconc newbody)))) (defmacro break () ' (return-from switch)))}
\]

Here we use two macros, one for switch and one for break, which together cause the statement

\[
\text{(switch n} \\
\text{ (0 (princ "none") (break))} \\
\text{ (1 (princ "one ")}) \\
\text{ (2 (princ "too ")}) \\
\text{ (3 (princ "many"))))}
\]

(which always prints either many, too many, one too many, none, or nothing) to expand into

\[
\text{(block switch} \\
\text{ (tagbody (case n} \\
\text{ (0 (go G0042))} \\
\text{ (1 (go G0043))} \\
\text{ (2 (go G0044))} \\
\text{ (3 (go G0045))) (return-from switch)} \\
\text{ G0042 (princ "none") (return-from switch) G0043 (princ "one ") G0044 (princ "too ") G0045 (princ "many")(return-from switch))}
\]

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 and 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 1960s and the 1970s and 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 1964], a version of COMIT with Lisp syntax. COMIT [MIT RLE 1962a, 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 of the early pattern-matching languages built on Lisp 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 1969, 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 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 1972a, 1972b], 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 1977d].) 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.

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 [Smith 1975]:

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...
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 coroutine 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. (For a more recent perspective, see [Hewitt 1991].)

Hewitt and his students (notably Russ Atkinson, Peter Bishop, Mike Freiling, Irene Greif, Roger Hale, Ken Kahn, Benjamin Kuipers, Todd Matson, Marilyn McLennan, Keith Nishihara, Howie Shrobe, Brian Smith, Richard Stieger, Kathy Van Sant, and Aki Yonizawa) 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; Smith 1975].

Although 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 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 \(\text{ct}\), \(\text{I}\), and \(\text{5}\). The XGP, at 200 dots per inch, made possible the printing of such exotic characters.) The recursive factorial function looked like this in PLASMA:

```
[\text{factorial} =
  \text{cases}
  (\rightarrow \{0\} 1)
  (\rightarrow \{\text{n}\} (\text{n} \ast (\text{factorial} (\text{n} - 1)))))]
```

Note the use of infix arithmetic operators. This was not merely clever syntax, but clever semantics; \((\text{n} - 1)\) really meant that a message containing the subtraction operator and the number 1 was to be sent to the number/actor/object named by \(\text{n}\).

One may argue that Lisp development at MIT took two distinct paths during the 1970s. 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 6.2.8); 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.
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 the 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 1975].

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.

Although 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; de Kleer 1978b] and truth maintenance systems [de Kleer 1978a; McAllester 1978] based on nonmonotonic 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 1980s 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 1980s, “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; 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 language 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. Although 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). S-1 Lisp [Brooks 1982a] was designed to use the multiple processors of an S-1 system, but (like so many other
features of S-1 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, 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. Thus pcall 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].


Qlisp [Gabriel 1984b; 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 indices. The idea was that indices are labels for virtual processors.

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.

6.5 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 six basic reasons: its theoretical foundations, its expressiveness, its malleability, its interactive and incremental nature, its operating system facilities, and the people who choose it.
Its theoretical foundations. Lisp was founded on the footing of recursive function theory and the theory of computability. The work on Scheme aligned it with Church's lambda calculus and denotational semantics. 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 expressiveness. Lisp has proved itself concerned more with expressiveness than anything else. We can see this more obliquely 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:

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

(defun add-matrix (ml m2)
  (let ((11 (length ml))
        (12 (length m2)))
    (let ((matrix (make-matrix 11 12)))
      (dotimes (i 11 matrix)
        (dotimes (j 12)
          (setf (nth i (nth j matrix))
                (+ (nth i (nth j ml))
                   (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. But this expectation is false for a sufficiently expressive language. When the primitives are at a sufficiently high level, there is enough wiggle room underneath to permit a choice of implementation strategies. Lisp implementors continue to explore that space of strategies. Precisely because Lisp is so expressive, it can be very hard to write fast programs (though it is easy to write pretty or clear ones).

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. 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 adventurous 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.

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. 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 five preceding factors merely serve to attract people to Lisp and 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 adventurous 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. (Recall that symbolic computation, now a field of its own, was at many institutions originally considered a branch of AI.) 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 languages other 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 inexpert 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, whereas most other languages are “fascist” (as hackers would have it [Raymond 1991]).

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.

This is one of the great advantages 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 [1985, 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 [1977]

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 Abelson [1985])

APL is like a beautiful diamond—flawless, beautifully symmetrical. But you can’t add anything to it. If you try to glue on another diamond, you don’t get a bigger diamond. Lisp is like a ball of mud. Add more and it’s still a ball of mud—it still looks like Lisp.

—Joel Moses [1978?]

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 Abelson [1985])

Lisp is the medium of choice for people who enjoy free style and flexibility.

—Gerald Jay Sussman (introduction to Friedman [1987], p. ix)

Hey, Quux: Let’s quit hacking this paper and hack Lisp instead!

—rpg (the final edit) [Gabriel 1992]
GUY L. STEELE JR. & RICHARD P. GABRIEL


[Fateman, 1973] Fateman, Richard J. Reply to an editorial. ACM SIGSAM Bulletin, 25, March 1973, pp. 9–11. This report 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.


[Gabriel, 1992] Gabriel, Richard P. Personal communication to Guy L. Steele, Jr., Nov. 30, 1992 (two hours before handing off this manuscript to Federal Express).


PAPER: THE EVOLUTION OF LISP


LISP SESSION

305


[Moses, 1978?] Moses, Joel, as recalled (and probably paraphrased) by Guy L. Steele, Jr. There has been a persistent confusion in the literature about this remark. Some have reported that Moses said it while on a panel at the ACM APL 79 Conference. Moses denies having ever made that particular remark, however, and indeed Steele has heard him deny it. Steele, however, is equally certain that Moses did make such a remark—not at the APL conference, but while standing in the doorway of Steele and White’s office, MIT room number NE43-834, circa 1978. Jon L White [personal communication to Steele, November 30, 1992] independently recalls having heard Moses comparing APL to a diamond and Lisp to a ball of mud on at least three separate occasions in that office building, once in NE43-834. The confusion will undoubtedly persist.


GUY L. STEELE JR. & RICHARD P. GABRIEL


STEELE: We’re very happy to be here this morning at the HOPL conference. We’ve worked very hard on this paper. We began work on it almost two years ago. So this has been a long time in the making and we’re ready to do it this morning.

GABRIEL: Yes, and we have also worked very hard on the presentation for this conference today. We’ve been working on the presentation for almost two days now. [laughter]

STEELE: This extreme range, a difference of orders of magnitude, is typical of various things that go on in the Lisp community. Let me give you a couple of examples. One is that there is an ANSI standard for the dialect of Lisp known as Scheme. It is a mere 50 pages long. It is a model of concision.

GABRIEL: The ANSI Common Lisp draft specification is at the opposite extreme, currently weighing in at 1,350 pages.

STEELE: That is a difference of 27 to 1. Another example is that with merely a stone knife, a block of wood, and a working C compiler, you can put together a working Lisp interpreter in less than a day if you know what you are doing.

GABRIEL: And, if you want to put together a fairly decent, but not outstanding, Common Lisp system, you should plan on 20 to 30 man-years. [laughter]

STEELE: We worked very hard putting this paper together. We polished it and polished it and got it as perfect as we could and there is no way we can present all of that material this morning in a space of 35 minutes. So we are not even going to try.

GABRIEL: So what we are going to do instead is try to take different slices through this complicated and very difficult to understand history of Lisp, and try to show you some themes that we’ve discovered over the years.

STEELE: We are assuming that you have read or probably will read the paper [in this book]. But that’s not a prerequisite for what we’re going to do this morning. Instead, we’re going to show you a graphical presentation of the various Lisp projects and we’re going to show you everything on this two-dimensional graphical plot [Figure 6.1], in which the vertical axis is time, extending from a little before 1960 to about the present. On the horizontal axis we’ve put geography running East and West. There is no reason why this should have turned out to be a useful axis, but it did, coincidentally, and so we’re going to lay things out that way. You will find over here on the left projects occurring in
Japan. There is a large cluster of stuff that happened in California; a variety of things that happened between the coasts in the U.S. at various scattered locations that we will talk about later—they are labeled by state on this plot; a whole pile of stuff that happened in Massachusetts, particularly centered around MIT where Lisp got started; and then there are a number of projects over in Europe.

GABRIEL: As often happens, you can learn quite a bit by plotting the relationships among the various languages and dialects. So we did that with the Lisp family graph and we came up with this intuitive look at the history of Lisp [Figure 6.2].

In this diagram the black arrows indicate a direct relationship, for example, either using the implementation to bootstrap a further one or to use part of the implementation later on. The magenta arrows [gray in this book], which took us three hours to draw in, represent an influence that is significant. Rather than show you all of this, we are going to show you different slices through this that illustrate some particular points.

STEELE: It would be helpful if I explained our color coding system. The red [lightest gray] rectangles represent specific Lisp implementation projects, and we have attempted to show the geographic location at which the implementation occurred, not necessarily the sites of their use. Where some of these bars are first wider, then narrow, we have intended the wide part to indicate roughly the time span of the heyday of those implementations, although use continued after their heyday for some time in some instances. The green [light gray] ovals indicate language specifications for languages that are not of the Lisp family but which we have included on the chart because they have an influence on the development of Lisp. The blue [darkest gray] ovals indicate language specifications within the Lisp family, but specifications as opposed to implementations. The cyan [dark gray] rectangles indicate official standards projects under ANSI or ISO.
There's no way in the world that we could possibly make intelligible sense out of this network of boxes and arrows. We’re going to show you different subsets of that. Let’s begin by looking at the Lisp 1.5 family [Figure 6.3].

This all stems from the original idea, shown here as a blue oval, of Lisp 1.5 as reflected in the *Lisp 1.5 Programmer's Manual* and earlier documents. This spawned a number of implementations. Perhaps the most direct descendants are the Lisp 1.5 implementations on IBM machines on the 7090 and the 7094 at MIT. There are a number of implementations early on on other machines, notably several different PDP-1 implementations that happened in different geographic locations; implementations on the M-460 and the Q-32, which you can read about in references that are cited in our paper; and also, notably, an implementation on the PDP-6, which was arguably the first real hardware Lisp machine. It was the first architecture that included instructions specifically for the support of the Lisp language and this turned out to have a great influence on the rest of the development of Lisp, as we will see later. There is also a version of Lisp developed in Cambridge, in England, and Lisp 1.5, after about ten or fifteen years, resulted in implementations in Utah (Standard Lisp) and the Stanford Lisp 1.6 and UCI Lisp configuration.

The most notable characteristic of this slide is that there is an important idea and suddenly there was a quick spawning of a bunch of implementations. Lisp was still a fairly small language, and at the time it was very easy for a project at another university or another company to say, “Hey, let’s build us a Lisp, too.” You could within a day, or a week, or a month, get some kind of Lisp working on whatever machine you had there locally.

**Gabriel:** Now we are going to look at a later family, the MacLisp family [Figure 6.4]. This family was developed here at MIT. Again, as Guy pointed out, it started off with the PDP-6 Lisp, which soon turned into MacLisp. MacLisp ran primarily on the PDP-10, which was again sort of a Lisp machine.
It had special instructions for doing \texttt{CAR} and \texttt{CDR} and good stack instructions for fast function call. This work was primarily done by a fellow named Jon L White, whom we see again and again in our story. He worked on this fairly continuously up until the beginning of the demise of MacLisp. As we can see by the direction of the arrows, MacLisp kind of spread; even though most people think of MacLisp as having remained at MIT, it moved across country, particularly when Scott Fahlman and Dave Touretzky moved from MIT to CMU. They, along with Guy Steele, did a port of it to the local system, where they did some extensions. I'm not sure what sort of adaptations they did, but they did some adaptations so that it became one of the standard Lisps at CMU.

Similarly, when I moved across country, MacLisp came with me and we did some significant adaptations to it on the West Coast. And, in particular, one thing you can see is that MacLisp was used for quite a long time at Stanford, way beyond where it was used at MIT. Other MacLisp-like dialects spawned off, for example, Lisp-Machine Lisp, which was kind of a response to the limited address space that the PDP-10 had. In fact, one of the significant events was running into the limit of basically a one-megabyte, one-MIPS machine. Keep in mind that people were trying to do AI research in this Lisp on this machine. There are several dialects of Lisp-Machine Lisp that we will see later.

One of the points we want to make here is a little bit surprising. We didn't know it until we saw these graphs. One of the things that you can see is that MacLisp kind of spread throughout the country and it did it for a couple of reasons. For one, MIT is relatively unstable in the sense that, when you are there, your funding may end at any time, so you might have to leave. So the tendency is for people to leave the area and take what they know with them. The, ah . . . I forget what the second point was!

\textbf{STEELE:} Instead of your second point, let me point out one block you missed: Richard Fateman left MIT when he went to Berkeley (as opposed to Stanford); the Franz Lisp effort got started, and this was another important offshoot of the MacLisp family.

I'll make a contrast with Dick's first point by looking at the IBM Lisp family [Figure 6.5]. Here there is a fairly direct relationship of taking at least the ideas, if not some of the implementation code, from the 7090 Lisp. Notice that this is the 7090 Lisp and not the 7094 Lisp—the difference between those is that the 7090 Lisp was running under the IBM operating system; the 7094 Lisp was running under MIT's CTSS operating system. It became Lisp360, and once it got there at Yorktown, or somewhere in New York, there was a fairly direct implementation line. It stayed in one place because IBM, at least compared to MIT, is a relatively stable place. Once you are there, you are set; there is less of a need to publish outside. There are more controls, for good reason: you have a product you are selling, as opposed to just doing it as an academic exercise. There are reasons to maintain central control over an implementation. Notice this is not the locale of use; this is the locale of implementation. There were, of course, users wherever IBM computers were.

There is one interesting connection here with Intedisp and with MacLisp, which is that there was a programming environment that—actually, Interlisp developed a great programming environment, and when Dick Gabriel happened to be at Illinois and he read about the Interlisp programming environment, he was greatly impressed and decided to bring a version of it up in MacLisp. He did that, and then when he went to Stanford, he carried that with him, brought it up there, and it became the programming environment of choice at Stanford, completely displacing the old Stanford Lisp 1.6. That programming environment then got ported back to the IBM environment and became the programming environment that was standard within VM Lisp. But except for that one strong outside influence, it was pretty much a self-contained series of projects there at IBM.

\textbf{GABRIEL:} Another example of a stable situation is with Interlisp [Figure 6.6]. Interlisp actually had its roots a little bit higher up in this diagram with the Basic PDP-1 Lisp that Peter Deutsch wrote as
a high school student in this area. This was later expanded into the SDS 940 Lisp, which was done, at least in part, by Danny Bobrow, who will, again, appear in many places in this story.

The interesting developments, from our point of view, start with BBN-Lisp. Now BBN-Lisp, which later became Interlisp, is interesting because it developed a notion of a very elaborate, cushy, comfortable programming environment. So, in contrast to MacLisp, which was concerned with performance and small size (in order to write large programs in the small address space of the PDP-10), Interlisp was more concerned about making the programming task easier, so there was a non-file-based structure editor, “Do What I Mean,” Conversational Lisp type of programming environment. At some point, BBN passed over the responsibility of maintaining BBN-Lisp to Xerox, partly coinciding with the departure of the principals, for example, Danny Bobrow, to the West Coast.

Interlisp has sort of a long history. As you can see, it doesn’t exist in very many places except for this box you can’t see, which is Interlisp 370, which was developed in Sweden, and I’m not sure how widely it was used. But if you ignore that, we can see a similar situation to IBM where there is sort of a narrow but long use. Again, because of the stability of the place, people didn’t tend to leave, they didn’t need to publish outside, so the ideas didn’t spread very much. There are a couple of interesting spawns here into LOOPS and CommonLoops, the Lisp Object-Oriented Programming System. We’ll see that later when we talk about the object-oriented programming aspects of this.

Danny Bobrow did quite a lot of work all over this map. One of the interesting things he did was a concept called spaghetti stacks, which were a kind of a coroutining facility, a way of saving state and going back and forth between different threads of control.

STEELE: Its important to realize that the Interlisp implementation, like the MacLisp implementation, was trying to run on a machine that had one megabyte of address space and ran at 1 MIPS; and that there were necessarily some severe engineering tradeoffs to be made. It’s not that the MacLisp group would not have liked to have had a good programming environment; it’s not as if the Interlisp group would not have liked to focus on compiler development and so forth, and good numerical performance, as the MacLisp group did; it is just that, when you are running under those kind of machine constraints, you have to make some hard choices. There was a time, in 1974, when the two implementation groups got together, at what we sometimes called a MacLisp-Interlisp summit, where the implementors got to agree on a set of features that they could exchange in order to encourage portability between the two platforms, and ultimately we agreed only on a few trivial things. At the time, I remember thinking, “Boy, those people are really pigheaded—they won’t take all our good ideas!” and I’m sure they felt very much the same way about us. Looking back on it, I’m inclined to the more generous interpretation that each of us, consciously or unconsciously thinking about our severe budgetary constraints of memory space and machine speed, simply felt that we couldn’t afford anything that wasn’t on our main program. But what broke this logjam was the development of the Lisp machines [Figure 6.7].

It became clear in the mid-70s that the PDP-10 wasn’t going to cut it for too much longer, and that it seemed to be more cost-effective to build custom-designed Lisp hardware—this was a point where you could just begin to afford to design special-purpose personal workstations. I think the original inspiration for this was the Alto at Xerox, although it wasn’t initially running Lisp. Then Tom Knight and Richard Greenblatt at MIT proposed to build these things called Lisp Machines, of which first a few dozen were built at MIT, and this then proceeded to spawn two companies, LMI and Symbolics, which then commercialized those. At the same time, Xerox produced a series of machines designed for both Lisp and Smalltalk, and some other languages, such as Mesa. So they had a series of personal workstations such as the Alto and the so called “D machines,” the Dorado, the Dolphin, and the Dandelion, and Interlisp was brought up on all those machines. There was also a machine that was
done at BBN, called the Jericho. The machine was geographically over here, but it used Interlisp, so I've notated it next to the box there for Interlisp. There was also some work done on Lisp machines in Japan, of which I think the most notable was the ELIS machine, which ran a dialect that was a combination of Lisp and Prolog, and this dialect was called Tao. The Lisp Machines lasted for quite a while and some of them are still in use today. There are Lisp Machines running over at Thinking Machines, doing diagnostic debugging of Connection Machines—that's just the software we happen to run on them right now. They are still considered workhorses in some places.

On the other hand, by the mid-80s, it became clear that machines that were custom-built to run just one language weren't necessarily cost-effective anymore, because the general-purpose workstation market had exploded and you could just go to Sun, or HP, or whatever place you liked, and buy a stock workstation, put up an implementation of Lisp on it, and the overall combination would be just as effective as having a custom-built piece of hardware.

GABRIEL: The next major development is Common Lisp [Figure 6.8]. Common Lisp is an interesting development in the sense that it is, I think, the first time that the Lisp community tried to work together. What happened was, basically, as you will recall from the MacLisp slide, all of the MacLisp people kind of split up, went to the four winds, and started working on their own various dialects of Lisp. Here are some of these dialects: NIL, ZetaLisp, Spice Lisp, S-1 Lisp are the primary ones. In April 1981, there was a DARPA meeting at SRI in California, sort of another Lisp summit meeting. The purpose of that meeting was for DARPA to find out whether it was going to be able to have a consolidated Lisp to use in its research. What happened at that meeting was that the Interlisp community, being sort of unified anyway, appeared very unified and the MacLisp community seemed kind of scatterbrained. So what we did was, we all got together and said, "Let's beat those guys." We got together and we started to define a description of a family of languages called Common Lisp.
Now Common Lisp has gone through a somewhat lengthy history. As mentioned, it takes quite a while to implement the Lisp. That’s because it’s a fairly large definition; a large, what most languages call library was included. This box here [points to (a) in Figure 6.8] represents the publication of CLTL1—Common Lisp: The Language, first edition—which represented the work of two or three years of informal standardization, which happened over the network, primarily. It was one of the first designs that happened over the network, and in fact, the e-mail traffic has been studied by various sociologists to figure out what was going on with us. [laughter] Later on, an ANSI standards committee was formed, called X3J13, to do a formal ANSI standardization of that. These boxes here represent the location of the chairman, Guy Steele, over here [points to (b)], and [puzzled] Guy Steele over there [points to (c)].

STEELE: [The diagram is] kinda crunched.

GABRIEL: [to audience] Okay, so we switched chairmen. The other interesting thing, which is not noted on here, is the extreme reaction against Common Lisp; we’ll come back to that later. [In fact, they didn’t.]

STEELE: Here is a picture of the Scheme extended family [Figure 6.9], and there are three points to be noted here. One is that this is a two-part story where Scheme represents a very sudden left-hand turn. Everything that happened before Scheme was a series of developments of a series of AI-oriented languages, actually going all the way back to COMIT, which was a very early string processing, pattern matching language at MIT. One of the notable offshoots of COMIT is that it had an influence on SNOBOL, but SNOBOL then did not have a lot of direct influence on Lisp; but it’s interesting to note it is a cousin of Scheme though a very distant connection. COMIT also spawned an implementation of COMIT in Lisp, which was called Meteor—there is an obvious pun there—that was done by Danny
Bobrow also, actually—and then after that there was a version of a pattern matching language called Convert, done by Adolfo Guzman at MIT, which was a redesign as a Lisp-embedded pattern matching language; it was used for some AI applications. Convert then had an influence on Carl Hewitt’s Planner, as described in his doctoral dissertation, and then Planner spawned a bunch of ideas, and what then began was what can be characterized as a debate, or an argument, between Carl Hewitt and Gerry Sussman and their respective students about how languages to support AI should be designed and how complicated control structure should be done. Sussman worked on the Muddle project, which was inspired by some of the ideas of Planner, and also on MicroPlanner and some others. Conniver was a response to Planner, suggesting that all the automated mechanisms of Planner that happened under the table to do coroutining and backtracking should be brought out and made explicit and put under the control of the programmer. And so Conniver was a coroutinist’s dream.

Scheme was originally called Schemer, and it was conceived of as another in the series of Planner and Conniver and so forth, and the name got truncated because file names on the PDP-10 were limited to six characters. [loud laughter and applause] And we decided that that was an okay name, too.

Scheme was an attempt to implement some of the actors ideas that Carl Hewitt had put in his language PLASMA, which had a very complicated and elaborate syntax. We were trying to strip it down to its barest essentials. After we had implemented a version of Lisp, we decided to give it lexical scoping as in ALGOL—there’s the ALGOL 60 influence—rather than the dynamic scoping that had been typical of Lisp 1.5, largely because Sussman had been studying ALGOL 60 and teaching it in a course and wanted to try that idea, too, and he thought it would work with actors.

When we got the whole thing implemented, we discovered that the code for implementing lexical closures and the code for implementing actors were identical—and this was discovered empirically—and we said, “Oh, I see! Actors are just closures of lambda expressions.” Well, there is a little
bit more to it than that, but suddenly we realized that Scheme was just a dialect of Lisp, rather than a complicated AI language. And at the point below that is the second part of the story, which is that Scheme, being a very small, simple version of Lisp, spawned a bunch of new dialects. There is a version at Yale called T, and there are a bunch of others that I haven't shown; there's work done at Indiana, by Dan Friedman and Dave Wise and their students, which was actually quite a hotbed of Scheme activity, and actually some Lisp activity even before Scheme got started—they influenced some of the early ideas of Scheme.

And we see a pattern here, hanging below Scheme, that is very much like the pattern that appears below Lisp 1.5, which is that a small simple idea suddenly could be implemented by a bunch of people and they could start to play with it, and we saw a large spawning of a bunch of implementations, which then got together and began to try to do some standards efforts. These green ovals, which are not on the master slide, show a series of revised Scheme reports—which are informal standards—and the third revision spawned the effort to produce the IEEE Scheme standard.

GABRIEL: The next thing we're going to see is the object-oriented influence, and this slide [Figure 6.10] is interesting for a couple of reasons. It's one of the few slides that show ideas and influence going one way and then the other, geographically. So, as we can see, one of the things that spawned all this off, of course, is the Simula 67 work, which influenced the Smalltalk work that we will hear about from Alan Kay later today. There is some question here about exactly how the Smalltalk work influenced the PLASMA and the Flavors work. The way that we interpreted it is that Smalltalk 71, as it's called, was the thing that influenced Carl Hewitt in a meeting that you can read about in our paper and in Alan's paper, where they got together and talked about the implementation of these sorts of languages. The later Smalltalk, which we call Smalltalk 72, had sort of a direct influence on Flavors;
several of the MIT folks had visited Xerox PARC, seen the idea, and said, "Yeah, we can do that a little bit better."

Now, one of the things this made us think about is, a lot of the other slides that you've seen have shown ideas originating here [at MIT] and spreading out. And we think that it's not so much because there's so many great ideas that come from MIT; rather, that MIT is a little more parochial, a little more inbred, and doesn't pay attention as much as the rest of the world to what's going on, so ideas tend to move away from here because ideas don't move in so much. [laughter] Sort of an extreme way of putting it. [laughter] The other thing is, MIT—well, you don't get a high salary, so people leave, and so they take their ideas with them. This is a contrast here, where we see the ideas coming back over.

Now, LOOPS, the Lisp Object-Oriented Programming System, is kind of an offshoot of Interlisp and is relatively Smalltalk-like. CommonLoops and CLOS [Gabriel consistently pronounced this "see-loss"], on the other hand, are very different. CommonLoops and CLOS attempt to merge the pseudo-functional programming style of Lisp with the object-oriented style, so that you have generic functions rather than message passing. In addition, from Flavors and New Flavors, CommonLoops and CLOS have multiple inheritance, also method combination, and a lot of other exotic object-oriented technology.

CLOS itself was an activity of X3J13, and I've listed a couple of other influences, namely Object Lisp and Common Objects. Object Lisp is probably a relatively direct descendent of actors—it's a delegation-based object-oriented system. And Common Objects, done by Alan Snyder, is sort of an encapsulation-heavy type of system. I have listed influences here, but they are negative influences. The CLOS group decided to explicitly ignore Object Lisp Common Objects. CLOS, as you can see, has influenced quite a lot of work, both here and in Europe. Dylan, ISLisp (which is the ISO standard Lisp), and EuLisp all use a variant of CLOS.

STEELE: Now I'd like to discuss the influence of the lambda calculus [Figure 6.11] on the development of Lisp. Those who have not studied Lisp very closely are aware that Lisp has lambda expressions, and that lambda calculus started that, and so forth. If you read McCarthy's paper in the first HOPL conference, he explains there that primarily what was borrowed from the lambda calculus was the lambda notation but not its precise semantics; rather, it was used in a slightly different way. Functions in Lisp 1.5 give dynamic scoping to their variables, rather than lexical scoping adhering to something like the ALGOL copy rule. So the notation came in there.

A second influence sort of focuses down here on Scheme and also on Lisp370, where through a series of influences—including Peter Landin's work on SECD machines and on ISWIM, and Evans' work on PAL, and Reynolds' work (particularly the "Definitional Interpreters" paper at the ACM '72 conference, which was a very strong influence on Sussman and me)—by these various intermediaries, Scheme and Lisp370 were led to implement lexical scoping, thereby preserving substitutability of expressions in a way that Lisp 1.5 did not.

The third influence of the lambda calculus is that, more recently, purely functional languages appeared, the so-called "functional languages," some of which are Lisp-like in syntax and some are not, but they are very clearly Lisp-like in spirit. I think the current culmination may be described as Haskell, which is a project that was done at Yale and other places, and that was in turn influenced by implementations of ML and Hope, which were in turn influenced by Landin. The influence coming in from lambda calculus is normal order reduction, that is, guaranteeing that if there is a normal form result for your computation, the interpreter will find it; this is not guaranteed in the case of Scheme or Lisp 1.5.
I guess the point to be made is that there is an interesting progression as Lisp moves onward, going back to its roots in lambda calculus, and discovering, in several cycles, there is more there in lambda calculus than we thought and the closer we get to lambda calculus, the better the language gets.

Gabriel: Moving from the sublime to the ridiculous, [laughter] the influence of FORTRAN [Figure 6.12] is relatively minor. This influence up here is merely that the first Lisp was the FORTRAN List Processing Language [FLPL], implemented in FORTRAN. The second influence is basically complex numbers. [For lack of time, Gabriel omitted mention of the influence of FORTRAN on FORMAT in Lisp-Machine Lisp.]

[to Steele] I guess we’ve got four minutes.

Steele: [to Gabriel] Right, okay.

[to audience] I’ve been on a number of standards committees where people say, “Well, this is an issue that can be decided in the FORTRAN community, because only FORTRAN worries about, say, complex numbers.” And I’d pop up and say, “Well, there’s also Common Lisp.” And [when they’d add] “Only FORTRAN has FORMAT,” I’d say, “Well, there’s also Common Lisp.”

Now we’d like to show you a series of peregrinations of some key individuals who were involved with Lisp projects. First we’d like to show you what happened with John McCarthy [Figure 6.13]. He worked at MIT for some time, and then in the mid-to-late-60s moved to Stanford. The important thing there is that he took PDP-1 Lisp with him, and this is how Lisp got started on the West Coast. That was a key migration, even if a short story.

Gabriel: The next guy is Danny Bobrow [Figure 6.14], who started here at MIT, worked on another PDP-1 Lisp, worked on a PDP-6 Lisp, worked on BBN-Lisp, and moved to the West Coast, where he
still is. This shows that he worked on UCI Lisp, spaghetti stacks, LOOPS, CommonLoops, and CLOS. He was involved in X3J13 and also worked on CLTL2.

Steele: There is another individual whose path parallels Danny Bobrow’s to some extent, which is Warren Teitelman, who also went from MIT and worked on Interlisp at Xerox; those two were the very strong influence from MIT on the Interlisp effort.

Here’s the path of Dick Gabriel [Figure 6.15], who got started on MacLisp at MIT, moved to Illinois—I’ve already discussed the influence of his Illinois presence, eventually, on IBM Lisp there—and from Illinois he went to Stanford and has been on the West Coast ever since then, except for a diversion for one summer to CMU, where he and I worked together on S-1 Lisp, which is shown here [in California] geographically, but actually, wherever we were, we were working on that. He was also involved in a large number of standards efforts, in Europe as well as in the U.S.

Gabriel: From the ridiculous to the sublime, Guy Steele. [laughter] Guy Steele [Figure 6.16] started here at MIT and then, through a series of trips out to the West Coast to work with me on S-1 Lisp, sort of bounced back and forth, and eventually went back to CMU and worked on Common Lisp. He was involved in a lot of the standards activities, worked on NIL, worked on Scheme, did a lot of stuff—but you know that.

Steele: Here are the travels of David Moon [Figure 6.17], and while he’s geographically been in Cambridge, he’s moved from project to project and carried a bunch of ideas back and forth. He got started working on Multics MacLisp. He was supporting the Multics implementation at the same time that I was working on the PDP-10 implementation. Then he was one of the principal implementors of the software for the Lisp Machine, and then moved over to Symbolics, and then has recently gone to Apple, where he’s been working on the Dylan project.
FIGURE 6.17

FIGURE 6.18
JOHN FODERARO

GABRIEL: Finally, there's Jon L White [Figure 6.18], who kind of disappears. He started here at MIT; he was the progenitor primarily of MacLisp. He took a trip to India [points to (a)] [laughter], came back [points to (b)] [laughter and applause] to work on MacLisp, went to IBM Yorktown to work on Lisp370, came back to work on MacLisp, went to Xerox to work on Interlisp, then left there and went to work for Lucid, where he works on Common Lisp. He has worked on all of these things—one of the few people who has switched among major families.

STEELE: Okay, do your sliding slot; I think we need to finish.

GABRIEL: [During this last bit, Gabriel took a piece of opaque paper with a narrow horizontal slot cut in it and moved it over the overhead for Figure 6.1 from top to bottom so as to reveal horizontal slices through the diagram, thereby converting the vertical axis representing time into an actual time axis.] One of the things that we've noticed is that if you look at the history as a little bit of a slice, you can see what things look like at different times [bell rings, indicating end of allotted time, but Gabriel is allowed to finish his sentence]—the number of implementations and dialects is scattered, kind of blooms in the middle of the 70s, starts to taper off, and ends up with a lot of standards activities (which never happened before) and only a very few dialects, showing that, possibly, the history is dying off. [applause]

TRANSCRIPT OF DISCUSSANT'S REMARKS

RANDY HUDSON: John Foderaro received a B.S. in Computer Science and Mathematics from the Pennsylvania State University in 1977. That year he began his graduate work at the University of California at Berkeley. His research focus was programming languages for symbolic algebraic systems. He worked on the design and implementation of Franz Lisp and on the design and implementation and testing of RISC in microprocessors. In 1983 he received his Ph.D. from Berkeley. The following year he co-founded Franz Inc. to support the Franz Lisp language and to implement Common Lisp.

JOHN FODERARO: Franz Lisp began in 1978 when DEC introduced the VAX minicomputer. I was at Berkeley at this point working with Richard Fateman on symbolic algebra systems. We had been using the MACSYMA system at MIT through the ARPANET. In fact, we had a very slow connection to the ARPANET, a 300-baud modem. So we wanted a version of Lisp and a version of MACSYMA on a local machine so we could actually do a little research on it.

The VAX was a great machine. It is a 32-bit flat-address-space machine, and it looked ideal for Lisp. At the time MIT had also received a VAX and we assumed they would come up with a Lisp and we would simply use their Lisp and get on with our work on symbolic algebra. Their Lisp was called NIL. As a stopgap we decided to work on Lisp anyway. It began as a port of Harvard Lisp. I am not sure where its origins are. Harvard Lisp ran on a PDP-11, written in an assembler language and was a very tiny interpreter in Lisp. A bunch of grad students took that, rewrote it in C, and brought it up on the VAX.

When it came time to give a name to this Lisp, we assumed this Lisp would be thrown away in a year so we were not thinking seriously about a name. Richard Fateman suggested the name Franz Lisp as a joke. We said, "It sounds pretty funny; let's use that name." We never thought the Lisp would ever leave Berkeley. It turned out that NIL never got out. I am not sure what its current state is now. It was lacking a garbage collector the last time I checked. NIL was more of an idea than a real implementation, probably because at MIT they had Lisp machines that were more interesting to play with than VAX's. So Franz Lisp actually was successful. A couple of names were not in the paper and
I want to mention their contribution: Keith Sklower was a staff programmer working for Richard Fateman. He did a lot of the work on the interpreter on Franz Lisp and the garbage collector and the bignum package and so on. Kevin Layer came on a little later. He was the first person to port Franz Lisp with another architecture. He ported it to a 68K and then went on to add a lot of Common-Lisp type features like packages and so on to it. They made significant contributions. There are other people throughout the world who added to Franz Lisp, too numerous to mention. But one I would like to mention is Tom London of Bell Labs who wrote the first compiler for Franz Lisp, a really excellent piece of work. My interest was actually in compilers. I actually wrote a few more compilers for Franz Lisp. That was my main focus.

I guess I should point out that Franz Lisp was actually successful. We ported MACSYMA to the VAX using it, basically adding MacLisp-like features to it. We were able to do a lot of work in symbolic algebra using Franz Lisp. When we first got our VAX it was only a half megabyte of memory and ran a swapping version of Unix, which came from Bell Labs. Both of these things meant it was really painful to use Lisp on this machine. It was a multi-user machine. We were trying to maybe share the machine with 20 people, and a half megabyte of memory doesn't give you much room. So we desperately needed a virtual memory, large address space Unix system. A couple of grad students at the university got interested in writing one, one of them being Bill Joy. They did a very good job of that, and as a result, this version of Unix, which was called Vmunix was distributed around the world. The everyday distribution of that Vmunix came with a set of tools from Berkeley, one of them being Franz Lisp. As a result of that, Franz Lisp ended up being installed as a standard piece of software on about 40,000 machines around the world. Contrast this to the other Lisp you saw in that diagram: most of them are either ones or twos, maybe they worked on one machine, maybe they were someone's idea of a Lisp, but this one actually got around. As a result of that, I think we helped propagate the idea of Lisp to a lot of people who never would have thought of using it. Even though on many of those machines, I would admit, people didn't actually use Lisp, on some of the machines, there were classes given. We estimate probably around 20,000 people actually used Franz Lisp throughout its lifetime—maybe more.

The paper mentioned that Franz Lisp was written entirely in C. Actually, it was written about 1/3 in C and 2/3 in Lisp, as typical for Lisp at that time. The fact that it was written in C is kind of unusual. At the time, most Lisp systems had an assembler language base. In fact, the NIL Lisp that was sort of our contemporary, I think, was designed with an assembler language kernel. We used C because that is what you did in Unix. C is the base for everything. I think nowadays people would sort of hesitate about starting out any large project written in an assembly language.

One particular feature of Franz Lisp that I want to mention is the foreign function interface. I think that is a major contribution and that its effects really have not been fully realized in the Lisp community. But I think it will be in the years to come. We had a symbolic algebra system, but we also did numerical work. We wanted access to large FORTRAN libraries, namely the IMSL libraries. We didn't want to have to decode them in Lisp, as people were doing, I understand, for Lisp machines. So we developed a system by which we could dynamically load in programs written in other languages, such as FORTRAN, C, or Pascal, and create automatic linkage functions from Lisp to those functions and call them just as if they were normal Lisp functions. We coined the phrase foreign function interface to describe this idea of talking to other languages. This has had a major effect on how I use the Lisp and how I think others have. First of all, no longer is the Lisp programmer kind of isolated. There are a lot more C programmers out there than there are Lisp programmers, and they are developing libraries, both math and graphic libraries. Lisp programmers used to have to sit by themselves and either recode the algorithms in Lisp or else just not use them. But with the foreign function interface people can actually load in C code well as the C programmer can. It is actually
TRANSCRIPT OF QUESTION AND ANSWER SESSION

easier for Lisp people to use C code than a C programmer because Lisp programmers can dynamically call out to the C functions through the interpreter. We could actually test C code and get back and try different values without having to recode the main program. So we had a really good connection to all the C functions. We could actually start with what the C programmer had and quickly build on it and generate nice environments.

The second effect is Lisp has always had a problem doing numerical work. Lisp is a great language for symbolic systems; it is a great language for doing object-oriented programming. It has the best object-oriented programming system now; CLOS is the most flexible but it has always had a problem doing numerical work because Lisp wants to keep track of all the types of all the objects at run time so we can do run-time dispatching, debugging, and so on.

There have been stabs at making good numerical compilers for Lisp but it requires the user to do lots of declarations and ensure that inside a loop he doesn’t do strange things that would cause values that have to be boxed and stored in the heap.

Through using the foreign function interface, users can now think of their applications in chunks. There is a numerical part, which they can code in their favorite numerical language be it C or FORTRAN, and the control portion, which they can code in Lisp. No longer does the Lisp compiler have to be large and massive and have great knowledge of great compilation. They can divide that portion off and let the C or FORTRAN compiler worry about that. Lisp systems, in particular Common Lisp, are really a huge language. I think needlessly so. Common Lisp has a huge numerical component. It is not a Common Lisp program unless it can do inverse hyperbolic arc tangent. This is just ridiculous. It was put in there because some people thought it might be interesting, but I bet not one person in ten thousand has ever used that for the reason it was there, other than to test on whether it works. That never should have been put in Common Lisp. If the designers of Common Lisp had gotten the idea that we should separate out what Lisp can do well from what FORTRAN and C do well, then we could let programs be built as a combination of the two languages. I think in the future Lisp will be used more as a control language for computational systems than as a whole complete language.

TRANSCRIPT OF QUESTION AND ANSWER SESSION

HUDDSON: We have time for some questions now. Our first question is from Alan Kay of Apple:

When and how did you realize how great Lisp is in the meta sense?

STEEL: I can report that I first cut my teeth on APL and thought that was the world’s most wonderful programming language. I still have a great deal of respect for it, and this is one of the reasons it had so much influence on the design on Common Lisp and some other dialects. I was convinced that Lisp was completely wrong-headed and made no sense at all. I tried to read the Lisp 1.5 manual and discovered it was trying to define the language in terms of itself; it was just not at all well-founded, and made no sense at all. A year later I had this sudden conversion, decided on an implementation, and then understood what the real idea was. So I would say that it was about 1972 when I suddenly realized how it hung together. It was difficult to glean from the 1.5 manual.

GABRIEL: My answer is the same, but with different details.

HUDDSON: (From Stavros Macrakis, OSF Research Institute): Earlier Lisps tend to have one dialect per architecture since interpreters weren’t portable. Ideas had to move by re-implementation. How have portable implementations changed the dynamics of innovation?

STEEL: I would say that innovation has decreased as portability has increased, but I think there is a chicken and egg problem there. I am not sure which is the effect.
GABRIEL: I think that people, now that they are trying to do portable systems, spend more time trying to make sure that they are implementing the same dialect, time that otherwise they would spend on great new ideas, so I think it has tended to slow things down. Also, standardization is freezing progress, and portability kind of implies standardization.

STEELE: A lot of the intellectual ferment in the Lisp community was with the InterLisp and MacLisp groups during the 70s. Once Common Lisp got started, that tended to freeze the MacLisp ideas, as you might expect. We saw the epicenter of this intellectual ferment and innovation transferred into the Scheme community. And Scheme proceeded to innovate in a number of ways, including the development of hygienic macros and some other things. Once the IEEE and then ANSI Scheme standardization efforts got going, that set of ideas tended to gel. I would say that now the locus of intellectual activity is probably within the Haskell group. This worries me a little bit, because while I think there are some really good ideas going on in Haskell, they have also abandoned some ideas that made Lisp very usable. I have been trying to use Haskell myself for the last year and have had mixed results. I have been very happy with some aspects of the language, particularly support for automatic currying. That is a much more expressive device than I thought I understood in the 70s. On the other hand, I found it very frustrating that I was trying to write program-transforming programs in Haskell and it doesn't come with a built-in parser the way Lisp does. So the first thing I had to do was to reinvent a bunch of tools in Haskell that I was used to having with the Lisp system before I could even get started. I am hoping that Haskell implementers will take that to heart.

HUDSON: (A question from Scott Guthrey of Schlumberger): A chart you didn't show is the use of Lisp in commercial systems. What would that chart look like?

STEELE: We actually thought of that chart and ran out of steam last night. That chart would look like this one, and you would see companies appear in the late 70s and early 80s and you would see more and more of them progressing towards the 90s. We are beginning to see some of them die off again in the computational marketplace. What are some of the ones that have disappeared, Dick?

GABRIEL: Teknowledge has kind of disappeared. They are down to a very small number. A whole raft of them disappeared.

STEELE: They came and they are going!

BIOGRAPHY OF GUY L. STEELE JR.

Guy L. Steele Jr. received his A.B. in applied mathematics from Harvard College (1975), and his S.M. and Ph.D. in computer science and artificial intelligence from M.I.T. (1977 and 1980). He has been an assistant professor of computer science at Carnegie-Mellon University; a member of technical staff at Tartan Laboratories in Pittsburgh, Pennsylvania; and a senior scientist at Thinking Machines Corporation.

He is author or coauthor of three books: Common Lisp: The Language (Digital Press); C: A Reference Manual (Prentice-Hall); and The Hacker's Dictionary (Harper&Row), which has been revised as The New Hacker's Dictionary, edited by Eric Raymond with introduction and illustrations by Guy Steele (MIT Press).

He has published more than two dozen papers on the subject of the Lisp language and Lisp implementation, including a series with Gerald Jay Sussman that defined the Scheme dialect of Lisp. One of these, "Multiprocessing Compactifying Garbage Collection," won first place in the ACM 1975 George E. Forsythe Student Paper Competition. Other papers published in are "Design of a
LISP–Based Microprocessor" with Gerald Jay Sussman (November 1980) and “Data Parallel Algorithms” with W. Daniel Hillis (December 1986). He has also published papers on other subjects, including compilers, parallel processing, and constraint languages. One song he composed has been published in the *Communications of the ACM* (“The Telnet Song,” April 1984).

The Association for Computing Machinery awarded him the 1988 Grace Murray Hopper Award. He was elected a Fellow of the American Association for Artificial Intelligence in 1990. He led the team that received a 1990 Gordon Bell Prize honorable mention for achieving the fastest speed to that date for a production application: 14.182 Gigaflops.

He has served on accredited standards committees X3J11 (C language) and X3J3 (FORTRAN) and is currently chairman of X3J13 (Common Lisp). He was also a member of the IEEE committee that produced the IEEE Standard for the Scheme Programming Language, IEEE Std 1178-1990. He represents Thinking Machines Corporation in the High Performance FORTRAN Forum.

He has served on Ph.D. thesis committees for seven students. He has served as program chair for the 1984 ACM Conference on Lisp and Functional Programming and for the 15th ACM Symposium on Principles of Programming Languages (1988); he also served on program committees for 30 other conferences. He served a five-year term on the ACM Turing Award committee, chairing it in 1990. He served a five-year term on the ACM Grace Murray Hopper Award committee, chairing it in 1992.

He has had chess problems published in *Chess Life and Review* and is a Life Member of the United States Chess Federation. He has sung in the bass section of the MIT Choral Society (John Oliver, conductor) and the Masterworks Chorale (Allen Lannom, conductor) as well as in choruses with the Pittsburgh Symphony Orchestra at Great Woods (Michael Tilson Thomas, conductor) and with the Boston Concert Opera (David Stockton, conductor). He has played the role of Lun Tha in *The King and I* and the title role in *Li’l Abner*. His “Crunchly” cartoons appear in *The New Hacker’s Dictionary*. He designed the original EMACS command set and was the first person to port TeX.

Richard P. Gabriel received his Ph.D. in Computer Science from Stanford University in 1981. He has been a researcher at Stanford University, a company president and Chief Technical Officer of Lucid, Inc., and a Consulting Professor of Computer Science at Stanford University.

He helped design and implement a variety of dialects of Lisp. He is author of one book, *Performance and Evaluation of Lisp Systems* (MIT Press). He has published more than 100 scientific, technical, and semi-popular papers, articles, and essays on computing.

He is the lead guitarist in a working rock ‘n’ roll band and a poet.