A Pattern of Language Evolution

Richard P. Gabriel
IBM Research
Guy L. Steele Jr.
Sun Laboratories

Preface

In 1992 when we completed our first draft of the History of Programming Languages II paper, *The Evolution of Lisp* [1], it included sections on a theory or model of how complex language families like Lisp grew and evolved, and in particular, how and when diversity would bloom and consolidation would prune. The historian who worked with all the HOPL II authors, Michael S. Mahoney, did not believe our theory was substantiated properly, so he recommended removing the material and sticking with the narrative of Lisp's evolution. We stopped working on those sections, but they remained in the original text sources but removed with conditionals.

Although the uncut version of the paper is published online [2], the theory was never officially published. This short paper is the publication of that material.

Pattern of Language Evolution

The evolution of Lisp since Lisp 1.5 [3] [4] is characterized by a cycle of diversification, acceptance, and consolidation. During diversification, new language constructs, new styles of programming, new implementation strategies, and new programming paradigms are experimented with and introduced to existing Lisp dialects, or new Lisp dialects are designed. In either case, a new Lisp dialect is, in effect, created. During acceptance, these new Lisp dialects are either accepted or rejected. The designers or backers of the new dialect will set conditions for acceptance, and the success of the acceptance phase will be determined by those conditions. During consolidation, a variety of dialects are merged. Typically one dialect will be chosen as the root and the branches will be taken from the same tree as the root or from other dialects. Consolidation is a process of standardization, either formal or informal. Consolidation, when most successful, results in a stable development platform.

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

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

Conditions for Acceptance

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

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

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

OOPSLA'08 October 19–23, 2008, Nashville, Tennessee, USA.

For example, Interlisp [5] and NIL [6] fell into disuse when their promoters chose the wrong computers.

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

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

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

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

Cachet does not always mean best of breed or most expensive or most exclusive. A 1967 Ford Mustang has more cachet than an 2008 Mercedes 300 SE, though the latter is far more expensive.

Acceptance groups comprise people who must individually wish to adopt a new dialect. The people who would use Lisp generally characterize themselves as leading edge. Therefore, there must be at least some aspects of leading edge technology in the new dialect. For example, the Common Lisp Object System (CLOS) [7] lends cachet to Common Lisp [8]—almost causing Common Lisp with CLOS to be regarded as a new dialect—because it is perceived as an advance over mainstream object-oriented programming.

Conditions for Consolidation

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

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

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

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

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

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

Conditions for Diversification

Diversification happens when old solutions are inadequate or when there is nothing else to do. The conditions for en-

tering diversification are that external driving factors are languishing, that there is a retreat to smaller research or development groups, and that the previous dialects have failed in some way.

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

The language experts will no longer be obligated to make these small improvements and will instead turn their attention to solving such problems as those which caused the acceptance group to decline. Perhaps a new acceptance group will be targeted along with its problems. Or perhaps another language or languages will have such cachet or success that the old language will be mined for incorporation into a new dialect.

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

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

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

Pollenation

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

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

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

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

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

The first real standard Lisps were MacLisp [9] and Interlisp.

ERRSET and CATCH

The lesson of ERRSET and CATCH is important.

Lisp 1.5 had a function called ERRSET, which was useful for controlled execution of code that might cause an error. The special form

`(ERRSET` *form*`)`

evaluates *form* in a context in which errors neither terminate the program nor enter the debugger. If *form* does not cause an error, ERRSET returns a singleton list of the value. If execution of form does cause an error, the ERRSET form quietly returns NIL.

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

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

that ERRSET could be reserved for its intended use of error trapping.

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

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

The next phase of MacLisp development began when the developers of MacLisp started to see a large and influential acceptance group emerge—Project MAC and the Mathlab/Macsyma [10] group. The emphasis turned to satisfying the needs of their user community rather than doing speculative / exploratory language design and implementation.

MacLisp

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

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

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

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

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

Thus MacLisp had an acceptance group, which had accepted MacLisp as its standard. Nevertheless, because MacLisp ran only on PDP-10s [11], there was little need to standardize the language through consolidation with other dialects.

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

The adoption of the Interlisp treatment of NIL was not received with universal warmth. We quote the public announcement by Jon L White:

> *For compatibility with Interlisp (foo), the CAR and CDR of NIL are always but always NIL. NIL still has a property list, and GET and PUTPROP still work on it, but NIL's property list is not its CDR (crock, crock). The claim is that one can write code such as (CADDR X) instead of the more time- and space-consuming (AND (CDR X) (CDDR X) (CADDR X)) and so on. Send complaints to GLS, but I doubt it will do you any good.*

A few words of explanation are in order. In MacLisp, CDR applied to an atomic symbol returned the symbol's property list. With this change, the symbol NIL becomes unlike all other symbols in there being no way to get its property

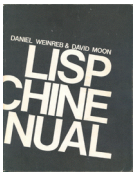
list. That CDR happened to work this way on symbols was an accident of implementation that users began to rely on heavily—this is another example of the unintended use of a feature that we saw with ERRSET and ERR. The contract for CDR, then, was not uniform: when applied to dotted pairs it did one thing, and when applied to symbols it did an abstractly unrelated thing that happened to be implemented by the same machine instruction. The solution was to complete the set of abstract operations for property lists; PLIST and SETPLIST were introduced to MacLisp in 1975, about nine months after the Interlisp compatibility change, so that CDR and RPLACD need not be used on symbols.

Documentation at that time was not exactly what we are used to today. The following were the only sources of information about MacLisp that users could reference:

1. The original PDP-6 Lisp manual and an update. [12]
2. `LISP ARCHIV` [13], a complete on-line log listing each newly added and deleted feature of the language. With each new release of MacLisp, release notes were prepended to the log. This file was maintained from 1969 until 1981, when Jon L White left MIT for Xerox. (The strange spelling is a consequence of the design of ITS, which limited file names to two components of at most 6 character each. On TOPS-10, with its 6-and-3 limits and different punctuation, the file was called `LISP.ARC`.)
3. The 1974 edition of the MacLisp Manual, written by David A. Moon as part of the effort to put MacLisp up on Multics. This valuable document was usually referred to orally as the "Moonual."
4. A short history of MacLisp published by Jon L White.
5. *The Revised MacLisp Manual* by Kent Pitman, written in 1983. [14]
6. "The wizard down the hall": implementors Jon L White and Guy Steele at MIT, and local MacLisp gurus such as Richard P. Gabriel at Stanford University, Rodney A. Brooks at Flinders University in Australia, David Touretzky at CMU, and Timothy Finin at the University of Illinois. Like the exiles in Ray Bradbury's *Fahrenheit 451*, these perhaps fanatical hackers were the documentation and would recite necessary information on request—a deplorable situation, perhaps, from today's perspective, but workable in a small community working with a rapidly changing piece of software.
7. The source code. Part of the spirit of the MacLisp community was the knowledge that when there was any doubt about what the language did, one could read the code.

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

The Interlisp effort, by contrast, produced a comprehensive reference manual for its more far-flung user community and a separate virtual machine specification that aided in porting Interlisp to different computer architectures. As the MacLisp community metamorphosed into the MIT Lisp Machine community, better documentation became a necessity and eventually appeared, thanks again to David Moon, this time with Daniel Weinreb. [15] (This one was called the "chine nual" (pronounced *sheenual*) because the title *Lisp Machine Manual* in big block letters was wrapped around the entire paperback cover so that only those letters showed on the front.)



Several other programming languages were developed within MacLisp: Micro-Planner [16], Conniver [17], and Scheme [18] to name a few. Several object-oriented extensions to MacLisp were tried out, including Extend and Flavors. Extend is a mechanism to extend the built-in Lisp type system using inheritance, and Flavors was a multiple inheritance, multi-method object-oriented addition to Lisp machine Lisp, which would become a major influence on the Common Lisp Object System (CLOS).

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 a severe constraint as the size of Lisp programs grew. MacLisp by this time had enjoyed nearly 10 years of strong use and acceptance within its somewhat small but very influential user community. Its implementation strategy of a large assembly language core would prove to be too difficult to work with for other computers, and intellectual pressures from other dialects, other languages, and the language design aspirations of its implementors would result in new directions for Lisp.

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

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

MacLisp versus Interlisp

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

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

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

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

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

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

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

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

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

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

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

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

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

This intermediate example is derived from a 1966 coding style:

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

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

With MacLisp, the style of indentation was consciously evolved, because programmers themselves were responsible for doing the indentation, at least until text editors like

Emacs added semi-automatic indentation facilities. Program code was viewed by text editors and on printout, while in Interlisp program code was primarily, but not exclusively, viewed on the screen.

Interlisp provided a pretty-printer that would take the in-memory program and dump it to a file or to the terminal. The structure editor used the pretty-printer to show program code. Though the pretty-printer was crudely programmable, few people customized it. Because MacLisp had a pretty-printer while programmers had developed their own pretty-printing style, the pretty-printer had to be programmable, and it was.

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

However, the primary mechanisms for language extensions were through the environment. DWIM (Do What I Mean), the spelling corrector, and CLISP (Conversational Lisp) served to define a new language by changing the surface syntax and tolerating certain types of errors. [5]

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

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

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

Minor Post-MacLisp/Interlisp Lisps

Standard Lisp [19] and Portable Standard Lisp (PSL) [20] represented a consolidation of ideas with the primary focus of delivering a particular program, Reduce [21], to a range of users who used among them a variety of Lisp dialects. The users of Reduce were on the rise and eager to use it. These two dialects were seen as necessary for the spread of the useful Reduce system. Standard Lisp was an attempt to piggyback on existing Lisps, while PSL represented an attempt to control performance a little better.

UCI Lisp [22] was an adaptation of Interlisp to a simple Lisp 1.6 [23] base, and so we can regard this as part of the consolidation of Interlisp.

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

Post-PDP-10

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

With the advent of the Digital Equipment Corporation Vax [24] this changed. There were no easy matches between Lisp data structures and Vax architectural facilities. Furthermore, the preferred Vax procedure call instructions did not fit well with the new style of MacLisp-like Lisp dialects: each of these dialects supported a variable number of arguments, and the Vax instruction set supported well only function calls with a fixed number of arguments, though there were several ways to cobble together an effective function call implementation for Lisp. At a time when other languages such as C and FORTRAN were learning to talk to each other, however, this provided yet one more impediment to Lisp's joining the crowd.

At the end of the 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. "Generalpurpose" workstations (i.e., workstations intended or designed to run languages other than Lisp) and personal computers hadn't quite appeared yet. To most Lisp implementors and users, the commercial hardware situation looked quite bleak.

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

At that point we had the situation where the natural acceptance groups for MacLisp were on the decline, but the acceptance groups of Interlisp were on the ascendency; these groups were the commercially minded AI groups that were about to be incorporated. Interlisp ran on a set of machines that were accepted by the commercial acceptance groups, and it fit their style of work. The new Lisp machines for Interlisp had a decided cachet. Things are ripe for Interlisp

to claim the right of consolidation, perhaps even winning over MacLisp users.

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

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

Notice that Franz Lisp [25] and PSL have nothing to offer the new acceptance groups except for portability, and each Lisp was a throwback to earlier MacLisp times. Though portability appears to be an important survival characteristic, without some cachet there was little hope for a dialect.

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

Sussman's work always had cachet (it seemed)—from Micro-Planner to Conniver to Scheme—and so his work almost always gained acceptance, but the acceptance groups did not have strong presence, so these ideas did not always catch on.

Surprising was the exception of MDL: Sussman belonged to this group but MDL [26] itself did not catch on, and at MIT it was not highly regarded. Nevertheless, a great many ideas from MDL caught on in Conniver and then Lisp-Machine Lisp. We still see these ideas hard at work in Common Lisp and to some extent in EuLisp. [27] Yet how many people who use these languages know of MDL, or its prime designer Chris Reeves?

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

In the global context, Scheme would provide concepts that would be used during the consolidation phases of other dialects. Scheme in the 1980s had one primary acceptance group, along with the potential of some others. The primary acceptance group for Scheme was the set of authors of the various revised reports on Scheme.

Early Lisp Machine History

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

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

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

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

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

Nevertheless, for most users the value lay ultimately in the software and not in its enabling hardware technology. The Lisp machine companies ran into difficulty in the late 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.

IBM Lisps

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.

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.

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

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

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

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

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

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

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

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

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

Common Lisp and CLOS

Common Lisp was the ultimate consolidation—taking ideas from just about all the post-MacLisp Lisps and some from Interlisp. Market forces dominate acceptance groups. The acceptance group for Common Lisp—commercial AI companies—went on the decline in 1989, and a flurry of diversification took place in terms of changes to Common Lisp, additions to Common Lisp, and EuLisp finally emerging (Europe was hit with AI winter first). Some parts of the acceptance group headed for Scheme because of size and simplicity.

Consolidation of Common Lisp caused Interlisp to die. Cachet of Lisp machines first and Suns later pushed Common Lisp ahead. The post-AI-winter era shrunk things down to small Lisp groups again, so, at that time, we expected a period of diversification.

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

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

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

Postface

The year 1993 was the last year covered in our history paper. Since then Lisp has gone into a bit of a slumber from which only in 2005 did it start to awaken. Of course it was never completely dead, but the world turned beginning in the mid-1990s to object-oriented programming in general and to Java in particular.

Many of the ideas in Lisp and in CLOS made their way into Java and more importantly into a raft of so-called "script-

ing languages" that followed on, such as Perl, Python, PHP, Javascript, and Ruby. If this can be taken as a diversification, the pattern we posited continues to this day.

References

[1] Steele, Guy L., Jr., Gabriel, Richard P., *The Evolution of Lisp*, Proceedings of the second ACM SIGPLAN conference on History of programming languages, 1993.

[2] http://dreamsongs.com/Files/HOPL2-Uncut.pdf

[3] McCarthy, John, Abrahams, Paul W., Edwards, Daniel J., Hart, Timothy P., Levin, Michael I., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[4] McCarthy, John, "History of LISP," In Wexelblat, Richard L., ed., *History of Programming Languages*, ACM Monograph Series, chapter IV, pp. 173–197, Academic Press, New York, 1981, ISBN 0-12-745040-8.

[5] Teitelman, Warren, et al., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, October 1978, Third revision.

[6] Burke, G. S., Carrette, G. J., and Eliot, C. R., *NIL Reference Manual*, Report MIT/LCS/TR-311, MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1983.

[7] Bobrow, Daniel, DeMichiel, Linda G., Gabriel, Richard P., Keene, Sonya, Kiczales, Gregor, Moon, David, *The Common Lisp Object System Specification*, Technical Document 88-002R of X3J13, LASC and SIGPLAN Notices, June 1988.

[8] Steele, Guy L., Jr., Fahlman, S. E., Gabriel, R. P., Moon, D. A., Weinreb, D. L., *Common Lisp: The Language*, Digital Press, Burlington, Massachusetts, 1984.

[9] Moon, David A., *MacLISP Reference Manual*, MIT Project MAC, Cambridge, Massachusetts, April 1974.

[10] Mathlab Group, *MACSYMA Reference Manual (Version Nine),* MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1977.

[11] Digital Equipment Corporation, Maynard, Massachusetts, *PDP-10 Reference Handbook*, 1969.

[12] Digital Equipment Corporation, Maynard, Massachusetts, *Programmed Data Processor–6 Handbook*, 1964.

[13] White, Jon L, et al, *LISP ARCHIV*, on-line archive of MacLisp release notes, 1969–1982.

[14] Pitman, Kent M., *The Revised MacLISP Manual*, MIT/LCS/TR 295, MIT Laboratory for Computer Science, Cambridge, Massachusetts, May 1983.

[15] Weinreb, Daniel L., Moon, David A., *LISP Machine Manual, Third Edition*, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1981.

[16] Sussman, Gerald Jay, Winograd, Terry, Charniak, Eugene, *Micro-PLANNER Reference Manual*, AI Memo 203A, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1971.

[17] Sussman, Gerald Jay, McDermott, Drew Vincent, *Why Conniving is Better than Planning*, AI Memo 255A, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, April 1972.

[18] Steele, Guy Lewis, Jr., Sussman, Gerald Jay, *The Revised Report on SCHEME: A Dialect of LISP*, AI Memo 452, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, January 1978.

[19] Marti, J., Hearn, A. C., Griss, M. L., Griss, C., *Standard Lisp report*, ACM SIGPLAN Notices, 14:10, pp. 48–68, October 1979.

[20] Utah Symbolic Computation Group, *The Portable Standard LISP Users Manual*, Technical Report TR-10, Department of Computer Science, University of Utah, Salt Lake City, January 1982.

[21] Hearn, A. C., *REDUCE 2: A system and language for algebraic manipulation*, Proc. Second Symposium on Symbolic and Algebraic Manipulation, pp. 128–133, Los Angeles, March 1971.

[22] Bobrow, Robert J., Burton, Richard R., Lewis, Daryle, *UCI-LISP Manual (An Extended Stanford LISP 1.6 System),* Information and Computer Science Technical Report 21, University of California, Irvine, Irvine, California, October 1972.

[23] *unknown*, *PDP-6 LISP (LISP 1.6),* AI Memo 116, MIT Project MAC, Cambridge, Massachusetts, January 1967; revised as Memo 116A, April 1967; the report does not bear the author's name, but Jeffrey P. Golden attributes it to Jon L White.

[24] Digital Equipment Corporation, Maynard, Massachusetts, *VAX Architecture Handbook*, 1981.

[25] Foderaro, J. K., Sklower, K. L., *The FRANZ Lisp Manual*, University of California, Berkeley, California, April 1982.

[26] Galley, S.W., Pfister, G., *The MDL Language*, Programming Technology Division Document SYS.11.01, MIT Project MAC, Cambridge, Massachusetts, November 1975.

[27] http://people.bath.ac.uk/masjap/EuLisp/