Lisp:
Good News
Bad News
How to Win Big

Richard P. Gabriel
Lucid, Inc.

January 4, 1994

Abstract

Lisp has done quite well over the last ten years: becoming nearly standardized, forming the basis of a commercial sector, achieving excellent performance, having good environments, able to deliver applications. Yet the Lisp community has failed to do as well as it could have. In this paper I look at the successes, the failures, and what to do next.

The Lisp world is in great shape: Ten years ago there was no standard Lisp; the most standard Lisp was InterLisp, which ran on PDP-10's and Xerox Lisp machines (some said it ran on Vaxes, but I think they exaggerated); the second most standard Lisp was MacLisp, which ran only on PDP-10's, but under the three most popular operating systems for that machine; the third most standard Lisp was Portable Standard Lisp, which ran on many machines, but very few people wanted to use it; the fourth most standard Lisp was Zetalisp, which ran on two varieties of Lisp machine; and the fifth most standard Lisp was Scheme, which ran on a few different kinds of machine, but very few people wanted to use it. By today's standards, each of these had poor or just barely acceptable performance, nonexistent or just barely satisfactory environments, nonexistent or poor integration with other languages and software, poor portability, poor acceptance, and poor commercial prospects.

Today there is Common Lisp (CL), which runs on all major machines, all major operating systems, and virtually in every country. Common Lisp is about to be standardized by ANSI, has good performance, is surrounded with good environments, and has good integration with other languages and software.

But, as a business, Lisp is considered to be in ill health. There are persistent—and sometimes true—rumors about the abandonment of Lisp as a vehicle for delivery of practical applications.

To some extent the problem is one of perception—there are simply better Lisp delivery solutions than are generally believed to exist—and to a disturbing extent the problem is one of unplaced or misplaced resources, of projects not undertaken, and of implementation strategies not activated.

Part of the problem stems from our very dear friends in the artificial intelligence (AI) business. AI has a number of good approaches to formalizing human knowledge and problem solving behavior. However, AI does not provide a panacea in any area of its applicability. Some early promoters of AI to the commercial world raised expectation levels too high. These expectations had to do with the effectiveness and deliverability of expert-system-based applications.

When these expectations were not met, some looked for scapegoats, which frequently were the Lisp companies, particularly when it came to deliverability. Of course, if the AI companies had any notion about what the market would eventually expect from delivered AI software, they never shared it with any Lisp companies I know about. I believe the attitude of the AI companies was that the Lisp companies will do what they need to survive, so why share customer lists and information with them?

Another part of the problem is the relatively bad press Lisp got, sometimes from very respectable publications. I saw an article in Forbes (October 16, 1989) entitled "Where Lisp Slipped" by Julie Pitta. However, the article was about Symbolics and its fortunes. The largest criticisms of Symbolics in the article are that Symbolics believed AI would take off and that Symbolics mistakenly pushed its view that proprietary hardware was the way to go for AI. There was nothing about Lisp in the article except the statement that it is a "somewhat obscure programming language used extensively in artificial intelligence."

It seems a pity for the Lisp business to take a bump partly because Julie thought she could make a cute title for her article out of the name "Lisp".

But, there are some real successes for Lisp, some problems, and some ways out of those problems.

1 Lisp's Successes

As I mentioned, Lisp is in better shape today than it ever has been. I want to review some Lisp success stories.

1.1 Standardization

A major success is that there is a standard Lisp—Common Lisp. Many observers today wish there were a simpler, smaller, cleaner Lisp that could be standardized, but the Lisp that we have today that is ready for standardization is Common Lisp. This isn't to say that a better Lisp could not be standardized later, and certainly there should be. Furthermore, like any language, Common Lisp should be improved and changed as needs change.

Common Lisp started as a grassroots effort in 1981 after an ARPA-sponsored meeting held at SRI to determine the future of Lisp. At that time there were a number of Lisps in the US being defined and implemented by former MIT folks: Greenblatt (LMI), Moon and Weinreb (Symbolics), Fahlman and Steele (CMU), White (MIT), and Gabriel and Steele (LLNL). The core of the Common Lisp committee came from this group. That core was Fahlman, Gabriel, Moon, Steele, and Weinreb, and Common Lisp was a coalescence of the Lisps these people cared about.

There were other Lisps that could have blended into Common Lisp, but they were not so clearly in the MacLisp tradition, and their proponents declined to actively participate in the effort because they predicted success for their own dialects over any *common lisp* that was defined by the grassroots effort. Among these Lisps were Scheme, Interlisp, Franz Lisp, Portable Standard Lisp, and Lisp370.

And outside the US there were major Lisp efforts, including Cambridge Lisp and Le-Lisp. The humble US grassroots effort did not seek membership from outside the US, and one can safely regard that as a mistake. Frankly, it never occurred to the Common Lisp group that this purely American effort would be of interest outside the US, because very few of the group saw a future in AI that would extend the needs for a standard Lisp beyond North America.

Common Lisp was defined and a book published in 1984 called "Common Lisp: the Language" (CLtL). And several companies sprang up to put Common Lisp on stock hardware to compete against the Lisp machine companies. Within four years, virtually every major computer company had a Common Lisp that it had either implemented itself or private-labeled from a Common Lisp company.

In 1986, X3J13 was formed to produce an ANSI version of Common Lisp. By then it was apparent that there were significant changes required to Common Lisp to clean up ambiguities and omissions, to add a condition system, and to define object-oriented extensions.

After several years it became clear that the process of standardization was not simple, even given a mature language with a good definition. The specification of the Common Lisp Object System (CLOS) alone took nearly two years and seven of the most talented members of X3J13.

It also became apparent that the interest in international Lisp standardization was growing. But there was no heir apparent to Common Lisp. Critics of Common Lisp, especially those outside the US, focused on Common Lisp's failures as a practical delivery vehicle.

In 1988, an international working group for the standardization of Lisp was formed. That group is called WG16. Two things are absolutely clear: The near-term standard Lisp is Common Lisp; a longer-term standard that goes beyond Common Lisp is desirable.

In 1988, the IEEE Scheme working group was formed to produce an IEEE and possibly an ANSI standard for Scheme. This group completed its work in 1990, and the relatively small and clean Scheme is a standard.

Currently, X3J13 is less than a year away from a draft standard for ANSI Common Lisp; WG16 is stalled because of international bickering; Scheme has been standardized by IEEE, but it is of limited commercial interest.

Common Lisp is in use internationally, and serves at least as a de facto standard until the always contentious Lisp community agrees to work together.

1.2 Good Performance

Common Lisp performs well. Most current implementations use modern compiler technology, in contrast to older Lisps, which used very primitive compiler techniques, even for the time. In terms of performance, anyone using a Common Lisp today on almost any computer can expect better performance than could be obtained on single-user PDP-10's or on single-user Lisp machines of mid-1980's vintage. Many Common Lisp implementations have multitasking and non-intrusive garbage collection—both regarded as impossible features on stock hardware ten years ago.

In fact, Common Lisp performs well on benchmarks compared to C. The following table shows the ratio of Lisp time and code size to C time and code size for three benchmarks.

	CPU Time	Code Size
Tak	0.90	1.21
Traverse	0.98	1.35
Lexer	1.07	1.48

Tak is a Gabriel benchmark that measures function calling and fixnum arithmetic. **Traverse** is a Gabriel benchmark that measures structure creation and access. **Lexer** is the tokenizer of a C compiler and measures dispatching and character manipulation.

These benchmarks were run on a Sun 3 in 1987 using the standard Sun C compiler using full optimization. The Lisp was not running a non-intrusive garbage collector.

1.3 Good Environments

It is arguable that modern programming environments come from the Lisp and AI tradition. The first bitmapped terminals (Stanford/MIT), the mouse pointing device (SRI), full-screen text editors (Stanford/MIT), and windowed environments (Xerox PARC) all came from laboratories engaged in AI research. Even today one can argue that the Symbolics programming environment represents the state of the art.

It is also arguable that the following development environment features originated in the Lisp world:

- Incremental compilation and loading
- Symbolic debuggers
- Data inspectors
- Source code level single stepping
- Help on builtin operators
- Window-based debugging

- Symbolic stack backtraces
- Structure editors

Today's Lisp environments are equal to the very best Lisp machine environments in the 1970's. Windowing, fancy editing, and good debugging are all commonplace. In some Lisp systems, significant attention has been paid to the software lifecycle through the use of source control facilities, automatic cross-referencing, and automatic testing.

1.4 Good Integration

Today Lisp code can coexist with C, Pascal, Fortran, etc. These languages can be invoked from Lisp and in general, these languages can then re-invoke Lisp. Such interfaces allow the programmer to pass Lisp data to foreign code, to pass foreign data to Lisp code, to manipulate foreign data from Lisp code, to manipulate Lisp data from foreign code, to dynamically load foreign programs, and to freely mix foreign and Lisp functions.

The facilities for this functionality are quite extensive and provide a means for mixing several different languages at once.

1.5 Object-oriented Programming

Lisp has the most powerful, comprehensive, and pervasively object-oriented extensions of any language. CLOS embodies features not found in any other object-oriented language. These include the following:

- Multiple inheritance
- Generic functions including multi-methods
- First-class classes
- First-class generic functions
- Metaclasses
- Method combination
- Initialization protocols

- Metaobject protocol
- Integration with Lisp types

It is likely that Common Lisp (with CLOS) will be the first standardized object-oriented programming language.

1.6 Delivery

It is possible to deliver applications written in Lisp. The currently available tools are good but are not yet ideal. These solutions include from removing unused code and data from application, building up applications using only the code and data needed, and producing `.o` files from Lisp code.

Delivery tools are commercially provided by Lucid, Franz, and Ibuki.

2 Lisp's Apparent Failures

> *Too many teardrops for one heart to be crying.*
> *Too many teardrops for one heart to carry on.*
> *You're way on top now, since you left me,*
> *Always laughing, way down at me.*
> ? & The Mysterians

This happy story, though, has a sad interlude, an interlude that might be attributed to the failure of AI to soar, but which probably has some other grains of truth that we must heed. The key problem with Lisp today stems from the tension between two opposing software philosophies. The two philosophies are called "The Right Thing" and "Worse is Better."

2.1 The Rise of "Worse is Better"

I and just about every designer of Common Lisp and CLOS has had extreme exposure to the MIT/Stanford style of design. The essence of this style can be captured by the phrase "the right thing." To such a designer it is important to get all of the following characteristics right:

- Simplicity—the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness—the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency—the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness—the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

I believe most people would agree that these are good characteristics. I will call the use of this philosophy of design the "MIT approach." Common Lisp (with CLOS) and Scheme represent the MIT approach to design and implementation.

The worse-is-better philosophy is only slightly different:

- Simplicity—the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
- Correctness—the design must be correct in all observable aspects. It is slightly better to be simple than correct.
- Consistency—the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness—the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

Early Unix and C are examples of the use of this school of design, and I will call the use of this design strategy the “New Jersey approach.” I have intentionally caricatured the worse-is-better philosophy to convince you that it is obviously a bad philosophy and that the New Jersey approach is a bad approach.

However, I believe that worse-is-better, even in its strawman form, has better survival characteristics than the-right-thing, and that the New Jersey approach when used for software is a better approach than the MIT approach.

Let me start out by retelling a story that shows that the MIT/New-Jersey distinction is valid and that proponents of each philosophy actually believe their philosophy is better.

Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the PC loser-ing problem. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine. It is called “PC loser-ing” because the PC is being coerced into “loser mode,” where “loser” is the affectionate name for “user” at MIT.

The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply try the system routine again. The MIT guy did not like this solution because it was not the right thing.

The New Jersey guy said that the Unix solution was right because the design philosophy of Unix was simplicity and that the right thing was too complex. Besides, programmers could easily insert this extra test and loop. The MIT guy pointed out that the implementation was simple but the interface to the functionality was complex. The New Jersey guy said that the right tradeoff has been selected in Unix—namely, implementation simplicity was more important than interface simplicity.

The MIT guy then muttered that sometimes it takes a tough man to make a tender chicken, but the New Jersey guy didn't understand (I'm not sure I do either).

Now I want to argue that worse-is-better is better. C is a programming language designed for writing Unix, and it was designed using the New Jersey approach. C is therefore a language for which it is easy to write a decent compiler, and it requires the programmer to write text that is easy for the compiler to interpret. Some have called C a fancy assembly language. Both early Unix and C compilers had simple structures, are easy to port, require few machine resources to run, and provide about 50%–80% of what you want from an operating system and programming language.

Half the computers that exist at any point are worse than median (smaller or slower). Unix and C work fine on them. The worse-is-better philosophy means that implementation simplicity has highest priority, which means Unix and C are easy to port on such machines. Therefore, one expects that if the 50% functionality Unix and C support is satisfactory, they will start to appear everywhere. And they have, haven't they?

Unix and C are the ultimate computer viruses.

A further benefit of the worse-is-better philosophy is that the programmer is conditioned to sacrifice some safety, convenience, and hassle to get good performance and modest resource use. Programs written using the New Jersey approach will work well both in small machines and large ones, and the code will be portable because it is written on top of a virus.

It is important to remember that the initial virus has to be basically good. If so, the viral spread is assured as long as it is portable. Once the virus has spread, there will be pressure to improve it, possibly by increasing its functionality closer to 90%, but users have already been conditioned to accept worse than the right thing. Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing. In concrete terms, even though Lisp compilers in 1987 were about as good as C compilers, there are many more compiler experts who want to make C compilers better than want to make Lisp compilers better.

The good news is that in 1995 we will have a good operating system and programming language; the bad news is that they will be Unix and C++.

There is a final benefit to worse-is-better. Because a New Jersey language and system are not really powerful enough to build complex monolithic software, large systems must be designed to reuse components. Therefore, a tradition of integration springs up.

How does the right thing stack up? There are two basic scenarios: the "big complex system scenario" and the "diamond-like jewel" scenario.

The "big complex system" scenario goes like this:

First, the right thing needs to be designed. Then its implementation needs to be designed. Finally it is implemented. Because it is the right thing, it has nearly 100% of desired functionality, and implementation simplicity was never a concern so it takes a long time to implement. It is large and complex. It requires complex tools to use properly. The last 20% takes 80% of the effort, and so the right thing takes a long time to get out, and it only runs satisfactorily on the most sophisticated hardware.

The "diamond-like jewel" scenario goes like this:

The right thing takes forever to design, but it is quite small at every point along the way. To implement it to run fast is either impossible or beyond the capabilities of most implementors.

The two scenarios correspond to Common Lisp and Scheme.

The first scenario is also the scenario for classic artificial intelligence software.

The right thing is frequently a monolithic piece of software, but for no reason other than that the right thing is often designed monolithically. That is, this characteristic is a happenstance.

The lesson to be learned from this is that it is often undesirable to go for the right thing first. It is better to get half of the right thing available so that it spreads like a virus. Once people are hooked on it, take the time to improve it to 90% of the right thing.

A wrong lesson is to take the parable literally and to conclude that C is the right vehicle for AI software. The 50% solution has to be basically right, and in this case it isn't.

But, one can conclude only that the Lisp community needs to seriously rethink its position on Lisp design. I will say more about this later.

2.2 Good Lisp Programming is Hard

Many Lisp enthusiasts believe that Lisp programming is easy. This is true up to a point. When real applications need to be delivered, the code needs to perform well. With C, programming is always difficult because the compiler requires so much description and there are so few data types. In Lisp it is very easy to write programs that perform very poorly; in C it is almost impossible to do that. The following examples of badly performing Lisp programs were all written by competent Lisp programmers while writing real applications that were intended for deployment. I find these quite sad.

2.2.1 Bad Declarations

This example is a mistake that is easy to make. The programmer here did not declare his arrays as fully as he could have. Therefore, each array access was about as slow as a function call when it should have been a few instructions. The original declaration was as follows:

```
(proclaim '(type (array fixnum *) *ar1* *ar2* *ar3*))
```

The three arrays happen to be of fixed size, which is reflected in the following correct declaration:

```
(proclaim '(type (simple-array fixnum (4)) *ar1*))
(proclaim '(type (simple-array fixnum (4 4)) *ar2*))
(proclaim '(type (simple-array fixnum (4 4 4)) *ar3*))
```

Altering the faulty declaration improved the performance of the entire system by 20%.

2.2.2 Poor Knowledge of the Implementation

The next example is where the implementation has not optimized a particular case of a general facility, and the programmer has used the general facility thinking it will be fast. Here five values are being returned in a situation where the order of side effects is critical:

```
(multiple-value-prog1
  (values (f1 x)
```

```
          (f2 y)
          (f3 y)
          (f4 y)
          (f5 y))
  (setf (aref ar1 i1) (f6 y))
  (f7 x y))
```

The implementation happens to optimize `multiple-value-prog1` for up to three return values, but the case of five values CONSes. The correct code follows:

```
(let ((x1 (f1 x))
      (x2 (f2 y))
      (x3 (f3 y))
      (x4 (f4 y))
      (x5 (f5 y)))
  (setf (aref ar1 i1) (f6 y))
  (f7 x y)
  (values x1 x2 x3 x4 x5))
```

There is no reason that a programmer should know that this rewrite is needed. On the other hand, finding that performance was not as expected should not have led the manager of the programmer in question to conclude, as he did, that Lisp was the wrong language.

2.2.3 Use of FORTRAN Idioms

Some Common Lisp compilers do not optimize the same way as others. The following expression is sometimes used:

```
(* -1 <form>)
```

when compilers often produce better code for this variant:

```
(- <form>)
```

Of course, the first is the Lisp analog of the FORTRAN idiom:

```
- -1*<form>
```

2.2.4 Totally Inappropriate Data Structures

Some might find this example hard to believe. This really occurred in some code I've seen:

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

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

What's worse is that in the particular application, the matrices were all fixed size, and matrix arithmetic would have been just as fast in Lisp as in FORTRAN.

This example is bitterly sad: The code is absolutely beautiful, but it adds matrices slowly. Therefore it is excellent prototype code and lousy production code. You know, you cannot write production code as bad as this in C.

2.3 Integration is God

In the worse-is-better world, integration is linking your `.o` files together, freely intercalling functions, and using the same basic data representations. You don't have a foreign loader, you don't coerce types across function-call boundaries, you don't make one language dominant, and you don't make the woes of your implementation technology impact the entire system.

The very best Lisp foreign functionality is simply a joke when faced with the above reality. Every item on the list can be addressed in a Lisp implementation. This is just not the way Lisp implementations have been done in the right thing world.

The virus lives while the complex organism is stillborn. Lisp must adapt, not the other way around. The right thing and 2 shillings will get you a cup of tea.

2.4 Non-Lisp Environments are Catching Up

This is hard to face up to. For example, most C environments—initially imitative of Lisp environments—are now pretty good. Current best C environments have the following:

- Symbolic debuggers
- Data inspectors
- Source code level single stepping
- Help on builtin operators
- Window-based debugging
- Symbolic stack backtraces
- Structure editors

And soon they will have incremental compilation and loading. These environments are easily extendible to other languages, with multilingual environments not far behind.

Though still the best, current Lisp environments have several prominent failures. First, they tend to be window-based but not well integrated. That is, related information is not represented so as to convey the relationship. A multitude of windows does not mean integration, and neither does being implemented in the same language and running in the same image. In fact, I believe no currently available Lisp environment has any serious amount of integration.

Second, they are not persistent. They seemed to be defined for a single login session. Files are used to keep persistent data—how 1960's.

Third, they are not multilingual even when foreign interfaces are available.

Fourth, they do not address the software lifecycle in any extensive way. Documentation, specifications, maintenance, testing, validation, modification, and customer support are all ignored.

Fifth, information is not brought to bear at the right times. The compiler is able to provide some information, but the environment should be able to generally know what is fully defined and what is partially defined. Performance monitoring should not be a chore.

Sixth, using the environment is difficult. There are too many things to know. It's just too hard to manage the mechanics.

Seventh, environments are not multi-user when almost all interesting software is now written in groups.

The real problem has been that almost no progress in Lisp environments has been made in the last 10 years.

3 How Lisp Can Win Big

> *When the sun comes up, I'll be on top.*
> *You're right down there looking up.*
> *On my way to come up here,*
> *I'm gonna see you waiting there.*
> *I'm on my way to get next to you.*
> *I know now that I'm gonna get there.*
> ? & The Mysterians

The gloomy interlude can have a happy ending.

3.1 Continue Standardization Progress

We need to bury our differences at the ISO level and realize that there is a short term need, which must be Common Lisp, and a long term need, which must address all the issues for practical applications.

We've seen that the right thing attitude has brought us a very large, complex-to-understand, and complex-to-implement Lisp—Common Lisp—that solves way too many problems. We need to move beyond Common Lisp for the future, but that does not imply giving up on Common Lisp now. We've seen it is possible to do delivery of applications, and I think it is possible to provide tools that make it easier to write applications for deployment. A lot of work has gone into getting Common Lisp to the point of a "right thing" in many ways, and there are viable commercial implementations. But we need to solve the delivery and integration problems in spades.

Earlier I characterized the MIT approach as often yielding stillborn results. To stop Common Lisp standardization now is equivalent to abortion, and that is equivalent to the Lisp community giving up on Lisp. If we want to adopt the New Jersey approach, it is wrong to give up on Lisp, because C just isn't the right language for AI.

It also simply is not possible to dump Common Lisp now, work on a new standard, and then standardize in a timely fashion. Common Lisp is all we have at the moment. No other dialect is ready for standardization.

Scheme is a smaller Lisp, but it also suffers from the MIT approach. It is too tight and not appropriate for large-scale software. At least Common Lisp has some facilities for that.

I think there should be an internationally recognized standard for Common Lisp. I don't see what is to be gained by aborting the Common Lisp effort today just because it happens to not be the best solution to a commercial problem. For those who believe Lisp is dead or dying, what does killing off Common Lisp achieve but to convince people that the Lisp community kills its own kind? I wish less effort would go into preventing Common Lisp from becoming a standard when it cannot hurt to have several Lisp standards.

On the other hand, there should be a strong effort towards the next generation of Lisp. The worst thing we can do is to stand still as a community, and that is what is happening.

All interested parties must step forward for the longer-term effort.

3.2 Retain the High Ground in Environments

I think there is a mistake in following an environment path that creates monolithic environments. It should be possible to use a variety of tools in an environment, and it should be possible for those who create new tools to be able to integrate them into the environment.

I believe that it is possible to build a tightly integrated environment that is built on an open architecture in which all tools, including language processors, are protocol-driven. I believe it is possible to create an environment that is multilingual and addresses the software lifecycle problem without imposing a particular software methodology on its users.

Our environments should not discriminate against non-Lisp programmers the way existing environments do. Lisp is not the center of the world.

3.3 Implement Correctly

Even though Common Lisp is not structured as a kernel plus libraries, it can be implemented that way. The kernel and library routines can be in the form of `.o` files for easy linking with other, possibly non-Lisp, modules; the implementation must make it possible to write, for example, small utility programs. It is also possible to piggyback on existing compilers, especially those that use common back ends. It is also possible to implement Lisp so that standard debuggers, possibly with extensions, can be made to work on Lisp code.

It might take time for developers of standard tools to agree to extend their tools to Lisp, but it certainly won't happen until our (exceptional) language is implemented more like ordinary ones.

3.4 Achieve Total Integration

I believe it is possible to implement a Lisp and surrounding environment which has no discrimination for or against any other language. It is possible using multilingual environments, clever representations of Lisp data, conservative garbage collection, and conventional calling protocols to make a completely integrated Lisp that has no demerits.

3.5 Make Lisp the Premier Prototyping Language

Lisp is still the best prototyping language. We need to push this forward. A multilingual environment could form the basis or infrastructure for a multilingual prototyping system. This means doing more research to find new ways to exploit Lisp's strengths and to introduce new ones.

Prototyping is the act of producing an initial implementation of a complex system. A prototype can be easily instrumented, monitored, and altered. Prototypes are often built from disparate parts that have been adapted to a new purpose. Descriptions of the construction of a prototype often involve statements about modifying the behavioral characteristics of an existing program. For example, suppose there exists a tree traversal program. The description of a prototype using this program might start out by saying something like "let S_1 be the sequence of leaf nodes visited by P on tree T_1 and S_2 the leaf nodes visited by P on tree T_2. Let C be a correspondence between S_1 and S_2 where $f : S_1 \rightarrow S_2$ maps elements to corresponding elements." Subsequent statements might manipulate the correspondence and use f. Once the definition of a leaf node is made explicit, this is a precise enough statement for a system to be able to modify the traversal routine to support the correspondence and f.

A language that describes the modification and control of an existing program can be termed a *program* language. Program languages be built on one or several underlying programming languages, and in fact can be implemented as part of the functionality of the prototyping environment. This view is built on the insight that an environment is a mechanism to assist a programmer in creating a working program, including preparing the source text. There is no necessary requirement that an environment be limited to working only with raw source text. As another example, some systems comprise several processes communicating through channels. The creation of this part of the system can be visual, with the final result produced by the environment being a set of source code in several languages, build scripts, link directives, and operating system calls. Because no single programming language encompasses the program language, one could call such a language an *epilanguage*.

3.6 The Next Lisp

I think there will be a next Lisp. This Lisp must be carefully designed, using the principles for success we saw in worse-is-better.

There should be a simple, easily implementable kernel to the Lisp. That kernel should be both more than Scheme—modules and macros—and less than Scheme—continuations remain an ugly stain on the otherwise clean manuscript of Scheme.

The kernel should emphasize implementational simplicity, but not at the expense of interface simplicity. Where one conflicts with the other, the capability should be left out of the kernel. One reason is so that the kernel can serve as an extension language for other systems, much as GnuEmacs uses a version of Lisp for defining Emacs macros.

Some aspects of the extreme dynamism of Common Lisp should be reexamined, or at least the tradeoffs reconsidered. For example, how often does a real program do this?

```
(defun f ...)

(dotimes (...)
  ...
  (setf (symbol-function 'f) #'(lambda ...))
  ...)
```

Implementations of the next Lisp should not be influenced by previous implementations to make this operation fast, especially at the expense of poor performance of all other function calls.

The language should be segmented into at least four layers:

1. The kernel language, which is small and simple to implement. In all cases, the need for dynamic redefinition should be re-examined to determine that support at this level is necessary. I believe nothing in the kernel need be dynamically redefinable.

2. A linguistic layer for fleshing out the language. This layer may have some implementational difficulties, and it will probably have dynamic aspects that are too expensive for the kernel but too important to leave out.

3. A library. Most of what is in Common Lisp would be in this layer.

4. Environmentally provided epilinguistic features.

In the first layer I include conditionals, function calling, all primitive data structures, macros, single values, and very basic object-oriented support.

In the second layer I include multiple values and more elaborate object-oriented support. The second layer is for difficult programming constructs that are too important to leave to environments to provide, but which have sufficient semantic consequences to warrant precise definition. Some forms of redefinition capabilities might reside here.

In the third layer I include sequence functions, the elaborate IO functions, and anything else that is simply implemented in the first and possibly the second layers. These functions should be linkable.

In the fourth layer I include those capabilities that an environment can and should provide, but which must be standardized. A typical example is `defmethod` from CLOS. In CLOS, generic functions are made of methods, each method applicable to certain classes. The first layer has a definition form for a complete generic function—that is, for a generic function along with all of its methods, defined in one place (which is how the layer 1 compiler wants to see it). There will also be means of associating a name with the generic function. However, while developing a system, classes will be defined in various places, and it makes sense to be able to see relevant (applicable) methods adjacent to these classes. `defmethod` is the construct to define methods, and `defmethod` forms can be placed anywhere amongst other definitional forms.

But methods are relevant to each class on which the method is specialized, and also to each subclass of those classes. So, where should the unique `defmethod` form be placed? The environment should allow the programmer to see the method definition in any or all of these places, while the real definition should be in some particular place. That place might as well be in the single generic function definition form, and it is up to the environment to show the `defmethod` equivalent near relevant classes when required, and to accept as input the source in the form of a `defmethod` (which it then places in the generic function definition).

We want to standardize the `defmethod` form, but it is a linguistic feature provided by the environment. Similarly, many uses of elaborate lambda-list syntax, such as keyword arguments, are examples of linguistic support that the environment can provide possibly by using color or other adjuncts to the text.

In fact, the area of function-function interfaces should be re-examined to see what sorts of argument naming schemes are needed and in which layer they need to be placed.

Finally, note that it might be that every layer 2 capability could be provided in a layer 1 implementation by an environment.

3.7 Help Application Writers Win

The Lisp community has too few application writers. The Lisp vendors need to make sure these application writers win. To do this requires that the parties involved be open about their problems and not adversarial. For example, when an expert system shell company finds problems, it should open up its source code to the Lisp vendor so that both can work towards the common goal of making a faster, smaller, more deliverable product. And the Lisp vendors should do the same.

The business leadership of the AI community seems to have adopted the worst caricature-like traits of business practice: secrecy, mistrust, run-up-the-score competitiveness. We are an industry that has enough common competitors without searching for them among our own ranks.

Sometimes the sun also rises.

References

[1] ? & the Mysterians, “96 Tears,” Pa-go-go Records 1966, re-released on Cameo Records, September 1966.