SPACE TELESCOPE SCIENCE INSTITUTE

Operated for NASA by AURA

The Planning and Scheduling Working Group Report on Programming Languages

John Michael Adams[*] Rob Hawkins Carey Myers
Chris Sontag Scott Speck Frank Tanner

April 4, 2003

[*]jmadams@stsci.edu

Contents

1 Introduction

This reports the results of a study undertaken at the Space Telescope Science Institute by the Planning and Scheduling Working Group (PSG), in order to evaluate the applicability of various programming languages to our astronomical planning and scheduling systems. While there is not universal agreement on all points, the available evidence supports the conclusion that Lisp was and continues to be the best programming language for the Institute's planning and scheduling software systems.

This report is structured as a sequence of articles. The first article constitutes the main body of the report and must be considered the official position of the working group. Subsequent articles are more detailed. These document individual positions taken by the committee members during the course of our study.

Concerning the relative merits of programming language and their applicability to specific projects, objective formal data are difficult to find. One may consult the ACM Computing Curricula [3] and language advocacy newsgroups to observe controversy at all levels of the computing community.

Given the difficulties, lack of unanimity is not unexpected. Our finding derives from the evidence and analysis supplied by the language subcommittee, synthesized under the guidance and approval of PSG.

Our finding is that Lisp best supports those aspects of system development most strongly applicable to the development, evolution and maintenance of our P&S systems. In support of the recommendation, we review the available evidence and analyze the known risks. Our analysis considers technical language features, technology trends, and business factors such as expected staffing costs and customer attitudes.

2 Methodology

We consented to limit our attention to Lisp, Java, Python, and C++. Our decision reflects judgments concerning adequate language power, market share, stability, and vendor independence. Certain nominally plausible candidates such as Smalltalk and Ruby were necessarily excluded to limit the candidate set to a practical size. C# was excluded as immature and unduly bound to the business interests of Microsoft and the Windows platform.

In an effort to probe and develop our perspective, we compiled a language feature comparison matrix. The matrix lists various language features and attempts to numerically describe the extent to which each languages supports that feature. Multiple inheritance and binary portability are examples of features. Non-technical factors were also considered. We performed a strawman exercise of assigning numeric importance factors to each of the features and then computing a scalar rank for each language on the basis of the weighted sum. As expected, the ranking exercise produced results according to individual taste and prior experience. Example matrices are included in section 12.

We dismissed Python the basis of anecdotal evidence of inadequate performance but also on grounds that, while excellent in many respects, it compares unfavorably to Lisp in expressiveness and Java in GUI support.

We dismissed C++ as too low-level (e.g., lack of native garbage collection)[1]. The group held that C or C++ might be reasonably applied to projects for which the performance requirements are particularly stringent. However, there is evidence suggesting that Lisp, as a natively compiled language, can deliver comparable performance. Thus, we determined that Java and Lisp were, by a large margin, the most natural candidates. Hence, most discussion was aimed at clarifying the relative merits of these two.

In the course of our work, participants were to individually develop, document and support a formal position, the goal being to document the important arguments in order for them to be effectively examined and synergistically developed. By management direction, we omitted the existing Lisp codebase from consideration. Clearly, the existing codebase is important for overall cost estimation, but this will be performed elsewhere. The individual positions appear in subsequent articles. These detail the basis of our recommendation. We summarize an important subset of the arguments in the remainder.

3 Developer Productivity

We find that Lisp affords a *considerable* productivity advantage over Java. Our finding is based on in-house experience and independent research.

The Lisp development environment offers numerous unique features. While bits and pieces of these are finding some level of support in developing Java areas such as the HotSpot VM, BeanShell, DynamicJava, and AspectJ, in comparison with Lisp facilities, they are crude and immature[2].

[1] Third party products such as Great Circle can offset this particular concern to some degree.

[2] AspectJ is anything but crude, but is immature in the sense that its technological relationship with Java proper has yet to be worked out. The author strongly encourages Java

3.1 The Lisp Top Level

Lisp includes an interactive facility called the top-level, a kind of Lisp shell, which allows one to type Lisp code for immediate evaluation. The facility is intended for exploration and debugging. It is not the mechanism one uses to express lengthy code fragments. One uses the top-level to inspect the state a the running program or to try out small pieces of code in order to understand API's or to prototype snippets prior to inclusion in application code. By providing persistent state and eliminating the load/restart cycle, the interactive facility accelerates the process of learning the language and also the production of application code.

BeanShell and DynamicJava are freeware Java tools that implement similar facilities. The author has explored these tools and found them disappointing. Since the Java language is not designed to be used interactively in a "scripty" way, these shells implement not Java, but Java-like languages. Moreover, they implement two different Java-like languages. Worse, they are not well integrated with development environments. Still, these tools are useful and good. The author recommends them to Java developers. However, they do not compare favorably to the analogous Lisp facilities.

3.2 Compiling Code into a Running Program

Lisp provides a high-level facility for adding and modifying code within a running program. Typically, one writes code into a file and then, from the editor, invokes the compile operation to integrate the current function or the current file with the running Lisp image. In this case, code is compiled to native machine language. This ability dramatically shortens the programming/test/debug cycle.

The savings is apparent even when prototyping a single function. Since one does not have to stop re-compile and re-load the application to test code changes, one remains focused on the task at hand and arrives more quickly at code with the desired behavior.

The savings is striking when working with a large application that requires a lengthy loading process for itself and/or its data. The Java development idiom is to write code, then restart the application and recreate state as needed for testing. With Lisp, one simply pops into the editor, changes the code, recompiles the changed code with a keystroke and then returns to the same application process—no application rebuild, reload, or state regeneration required.

developers to study AspectJ.

The Java HotSpot VM is a step in this direction. The HotSpot VM is an evolved Java VM with special debugging support for loading class files into the application under debugger control. The Sun Microsystems white paper explains that the motivation for HotSpot is a desire for better platform-independent debugging facilities. It is not yet clear the extent to which it supports rapid, incremental development.

3.3 Lisp is its own Debugger

Unless explicitly excluded, facilities to compile, load and debug Lisp code are always present in the application image. Thus, *Lisp is its own debugger.* When debugging Lisp programs, one has full, unfettered access to the entire language and all of its API's and other facilities. This makes it easy to write ad hoc diagnostic and exploratory code and to prototype solutions on the fly.

3.4 Fix-and-restart

The crowning result of all the above is that one can fix Lisp programs while they are running, even if an exception such as divide by zero occurs. When this happens, the Lisp environment enters the debugger[3]. One can, as with any debugger, inspect the stack and the various contexts to diagnose the problem. Upon diagnosis, one can compile the fix directly into the running program and restart the program *from the point of exception.*

Contrast this with Java development. When an uncaught exception occurs, even under debugger control, the program will terminate. The program will have to be restarted with a breakpoint to inspect the program state prior to the exception in order to diagnose the problem. When the code is fixed, the program will have to be restarted *again* to verify.

Our P&S systems often involve processing that runs into tens of minutes and even hours. By forcing a restart of these long runs once or twice in every debug cycle, the Java model imposes a prohibitive tax on development effort.

3.5 Verification Support

These above described benefits apply to testers as well as developers. When a tester is working with an application, and an error occurs, the developer must

[3]This behavior is configurable. Production applications can be made to exit or take arbitrary actions in preference to entering the debugger and waiting for developer input.

usually become involved in the diagnosis of the problem. With Lisp, since the development environment can be present in the application image, diagnosis can begin immediately in the identical context of the tester's observation.

With most other languages, and certainly Java, the tester will have to describe how to reproduce the problem and the developer will have to do this in a new process under debugger control. But this is problematic, since the problem may be difficult to reproduce, even with identical input. Or, the problem may have occurred following a complex sequence of events that cannot precisely be recalled by the tester.

To this we can add that Lisp provides excellent support for remote debugging. It does this in two ways. If a tester encounters a problem, it is possible, for the developer to bring up the Lisp display on a different X display or even over the Internet. It is also trivial to setup a telnet server with the Lisp environment so that a developer can connect to the Lisp process via telnet. This is helpful when Lisp is not attached to X.

4 Performance

Although all Lisp environments come with interactive facilities and many of these include an interpreter, Lisp compilers generate *native machine code*. Studies have shown that the performance of Lisp is superior to Java and is comparable to C++ in some cases.

Performance measurements are usually of limited value beyond the specific application and platform context in which they are made. In general, we held that performance studies were out of scope, owing in large measure to a lack of detailed requirements. Various studies are found in the literature. Most of these focus on floating point intensive applications. In some cases, Lisp can be made to run as fast or faster than the equivalent C or Fortran programs. In other cases not.

As a rule, the fastest C++ programs will be faster than their Lisp counterparts, which will in turn be faster than the Java. C/C++ are statically typed and natively compiled. Lisp is natively compiled but dynamically typed, so that runtime type-checking imposes a performance tax. Java is statically typed but interpreted. Note that Lisp, while dynamically typed, permits optional type declarations which can be leveraged by the optimizing compiler to produce arithmetic code which is comparable to statically typed languages. If the choice is between Java and Lisp, Lisp has the performance edge.

More importantly, accepted principles of optimization favor Lisp over C/C++

for most applications. First, it is known that performance bottlenecks occur in a small minority of code. Second, when one appears, the problem is just as likely, if not more so, to arise from defects in data structures and algorithms than in the compiler-generated code. In other cases, optimizing Lisp via type declarations will sometimes solve the problem. For the remainder, it may be necessary to code in a lower-level language, a scenario to which Java is expected to be more prone since it is interpreted by design[4].

It is not clear that interpreted Java can supply the needed performance for P&S applications. However, it is reasonable to expect that this will become less of a concern over time.

5 GUI Technology

We find at the present time that Java's Swing API is the superior multi-platform GUI facility. Until a better alternative emerges, most GUI code should be written to the Swing API. With Lisp as the backend, this implies a multi-language approach. A multi-language approach has disadvantages which are discussed in 8.2.

GUI technologies have changed rapidly throughout their history and can be expected to change in the future. There is no reason to believe that a Java-based Swing API will be the GUI standard fives years from now. It is even less likely that Java is the final word in programming language technology. Attractive new languages will emerge and will also need GUI support. Thus, the predominate technological pressure will be toward language independent GUI services rather than programming language homogeneity for the sake of particular API's.

6 Planning and Scheduling Considerations

Several factors pertaining specifically to our historical planning and scheduling development work support Lisp. Historically, our users and developers have found it necessary to continually evolve our approaches to planning and scheduling. Lisp's superior support for prototyping, its interactive facilities and its uncommon expressiveness far outstrip Java's ability to support system evolution.

The ability to compile changes into a running program is considered critical. We frequently have had occasion to apply search algorithms to sizable datasets.

[4]Java's JIT (just in time compilation) may be helpful in some cases, but it is not apparent that Java is presently deserving of natively compiled status.

In the absence of the ability of Lisp to modify the system as it runs, many development efforts would be extended by days or even weeks by the necessity to restart long runs for every change in the code. This critically impedes system evolution and time to delivery.

6.1 Integrative vs. Domain Logic

It is sometimes useful to conceptually separate the features of a system into the categories of *integrative* and *domain* logic. The domain logic is that part of the system structure which maps directly into the application domain. The integrative logic is that part of the system structure which is accidental to current technology or the larger system environment in which the application must function. For example, code that must submit an SQL query to read a database is integrative code. Its structure has no meaningful relationship to the application domain. On the other hand, code that say, defines a data structure to represent visibility periods for astronomical scheduling, we consider domain logic.

This distinction can guide technology selection. While applications usually have some of both kinds of code, some applications are dominated by one or the other. Consider a data-entry application that reads and writes records to a database. The key concepts such as menus, data-entry fields, database tables, etc., are so common as to support huge vendor markets for GUI and database services. The application is simple, its requirements well understood having been implemented countless times and in countless places. The building of such an application is not an occasion for deep thought, but rather an occasion for familiarity with the most useful and stable API's. Given that familiarity, and the common nature of the requirements, one can expect to build the application by integrating vendor components and without remarkable innovation in algorithms, architecture or design. For applications that tend toward this end of the spectrum, it is only natural to pick whatever language provides the best interfaces to these common services. The language itself matters less since the programming job consists mostly in making API calls to vendor supplied routines. For smaller, integration-bound applications, Java is likely to be the preferred vehicle.

On the other hand, if an application is domain-bound, meaning its structure is determined mostly by complex, domain specific modeling, then it is more important to bias language selection toward those which can best express complex data structures and algorithms, while offering the most powerful features for incremental, iterative and exploratory programming. It is far from a common requirement to say, compute visibility periods for an astronomical observation. Thus, it is not to be expected that we can build our P&S applications out of readily available API's. There are two countervailing economic considerations.

First, planning and scheduling, within and without the field of astronomy, continues to be a field of active research. People simply do not know how to write generalized services that scale properly and are readily portable across domain boundaries. Second, the potential market for these kinds of services is small compared to other areas such as GUI services or enterprise middleware.

Our applications are clearly domain-bound. Our development work is dominated by novel conceptual concerns arising from the unusual nature of our science mission. For this type of application, we find as have many others in diverse fields, that Lisp is *greatly superior to Java* because it is more expressive and better supports a fluid, evolutionary approach to development.

7 Supporting Research

7.1 The Work of Capers Jones

In Applied Software Measurement [2], Capers Jones, the noted software engineering and measurement expert, quantifies the unit costs of writing software by estimating number of source code statements required per function point for many programming languages. [5]

The number given for Common Lisp is 21; for Java, 53. It is expected that Java programs will tend to be approximately twice the size of the equivalent Lisp programs. Other things being equal, this translates to higher development and maintenance costs for Java programs.

7.2 Prechelt and Gat

Erann Gat of JPL, extending a study by Prechelt[4] studied development time, program length, runtime and memory consumption between Lisp, Java and C/C++ programs[1]. His study found that Lisp programs were significantly shorter and required less time to develop. Moreover, the time required to develop the Lisp programs was significantly less variable. Gat writes,

> Development time for the Lisp programs was significantly lower than the development time for C, C++, and Java programs. It was also significantly less variable... Programmer experience cannot account for the

[5]This table appears on page 76 of the first edition. Published in 1991, this edition does not include relevant data on Java and Common Lisp. An updated table including entries for Java and Lisp (the relevant entry is actually CLOS, which includes the object-oriented facilities of Common Lisp) can be found at http://www.theadvisors.com/langcomparison.htm .

> difference. The experience level was lower for Lisp programmers than for both the other groups (an average of 6.2 years for Lisp versus 9.6 for C and C++ and 7.7 for Java). The Lisp programs were also significantly shorter than the C, C++, and Java programs.
>
> Lisp programs ranged from 51 to 182 lines of code. The mean was 119, the median was 134, and the standard deviation was 10. The C, C++ and Java programs ranged from 107 to 614 lines, with a median of 244 and a mean of 277.
>
> Although execution times of the fastest C and C++ programs were faster than the fastest Lisp programs, the runtime performance of the Lisp programs in the aggregate was substantially better than C and C++ (and vastly better than Java). The median runtime for Lisp was 30 seconds versus 54 for C and C++. The mean runtime was 41 seconds versus 165 for C and C++.[1]

Gat's study extends the Prechelt study, which did not consider Lisp.[6] His sample consists of 16 programs written by self-selected participants. While not conclusive, the study is suggestive and confirms the independently derived Capers Jones numbers. These studies are the best independently derived formal data known to exist comparing Lisp with other languages.

8 Risks

8.1 Staffing Risk

Some feel that the smaller Lisp market poses, a priori, a hiring risk. Such concerns are usually founded upon the assumption that we ought to limit our hiring to those with significant prior experience in the language of the application or else face prohibitive staffing costs. The experience of PSDB and its organizational predecessors is otherwise. Certainly, there are thresholds of complexity and project duration below which it makes no sense to apply other than the most widely known languages. However, there are just as clearly similar thresholds above which an investment in training in a technically superior tool will pay off.

We have a decade of experience with this approach, having used it to produce an acclaimed application of unprecedented applicability to astronomical planning. We have experienced no difficulty in attracting highly qualified developers. Moreover, our substantial experience with having developers learn Lisp on the job is positive. With only a few days of study and some support from the more experienced, our programmers achieve competency in the language in

[6]Thus, the programming problem cannot have been chosen to favor Lisp.

a few days or weeks. By contrast, it takes months or years to come up to speed in the application domain of HST P&S. In our particular field, understanding that productive language competency comes in a matter of weeks, we view as sorely misguided any hiring policy that emphasizes specific language experience in preference to insightfulness and technical maturity.

Thus there are two important considerations that counter this fear of staffing risk. The first is that our application domain is sufficiently complex that total training costs will be dominated by the domain itself, so that the choice of language is negligible with respect to the total cost of producing "up to speed" programmers. Second, our application domain is sufficiently novel that we have a much greater need for the expressiveness and productivity support of Lisp than for the commodity API's of Java, and thus we should be teaching Lisp as needed in preference to hiring Java programmers.

8.2 Multiple Languages

If Lisp is to be used for future Spike development[7], it is probable that Java would be chosen for the front-end in order to code to the Swing API. While not ideal, we consider this an appropriate application of the principle of choosing the best tool for the job. We foresee little risk of significant integration costs arising from having to talk between languages or processes to implement the strategy.

Nearly all projects beyond a certain scale, end up as multi-language projects either because attractive alternatives to the original language emerge or else because the requirements of the application demand the best features of more than one language. Efforts to maintain a single-language solution for its own sake may inhibit important innovations.

We also note an ancillary benefit. It is a widely recognized principle of programming that user interface code should be clearly distinguished and isolated from the core logic of an application. But this principle is rarely followed with the consistency one might expect from the earnestness of its frequent expression. Having a separate process for the GUI gives firm expression to this doctrine and can also permit greater flexibility in testing. In addition, the enforced separation allows multiple interfaces to exist in isolation from the core code, which can assist in technology migration and innovation.

[7]Spike is a Lisp application used for long-range planning of Hubble Space Telescope observations.

8.3 Technology Trends

Contrary to popular misconception, there has been a trend over the past two decades for Lisp to move out of computer science laboratories and into the competitive commercial arena. A surprising number of commercial applications have been recently developed in Common Lisp.

ITA software supplies the fare search backend to the popular Orbitz website (and others). The nature of pricing in the commercial airline industry involves a combinatorial explosion demanding of high performance. At the 2002 International Lisp Conference, in addition to noting the high performance potential, ITA engineers emphasized the criticality of Lisp's ability to succinctly describe complex algorithms and also, given their need to work with gigabyte sized datasets, the aforementioned ability to "fix and restart".

The popular Sony Playstation game, Jak and Dexter provides another contemporary example of commercial success enabled by Common Lisp. Naughty Dog software used Lisp in the creation of its gaming development environment. More example of applications can be found at the Franz, Inc.[8] website.

These examples demonstrate not only that Lisp applications can deliver high-performance, but that, contrary to popular misconception, Lisp underlies many remarkable and contemporary commercial successes.

8.4 Vendor Support

We know of four commercial Lisp vendors, at least three of which have been in business for over a decade, in some cases two. There is little risk of the disappearance of high quality vendor support. Franz, Inc. has given consistently high quality service over the years. The smaller Lisp market space works to our advantage here. It is much easier to give good service to 2000 customers than 200,000.

In addition to commercial offerings, there are numerous, high-quality, actively maintained free implementations including CMUCL, SBCL, CLisp, and OpenMCL.

[8]The Institute's Lisp Vendor. www.franz.com

8.5 Stability

Owing to the rapid evolution of Java technologies, Lisp environments exhibit considerably higher stability while continuing to add support for important technologies such as CORBA and XML. It must be admitted, however, that the relative instability of Java technology is due to rapid innovation—a good thing. It is to be expected that as new protocols and data formats emerge, native Lisp support will sometimes lag behind Java. Because our applications are domain-bound, we do not consider this a critical consideration in our case. Nevertheless, we can add that commercial Lisp Vendors have many Fortune 500 clients and hence a strong motivation to keep up to date with emerging IT trends. Moreover, as noted later, Lisp offers good support for calling C and Java based API's which in many cases can be expected to permit rapid access to emerging API's where deemed critical.

9 Customer Attitudes

For various reasons, the capabilities and current technological condition of Lisp are not widely understood. Those who know anything about Lisp, typically have perceptions based on knowledge or experience that is ten or twenty years outdated. For example, it is commonly believed, erroneously, that Lisp is not natively compiled and therefore slow. Another common belief is that Lisp is a niche language. Outdated textbooks and university teachers perpetuate some of these misconceptions.

Customers and potential customers cannot be expected to escape such misconceptions. Thus, customers may have concerns or objections to the use of Lisp. This is an important matter.

Surely, if Lisp is expected to deliver higher quality systems at a lower cost, we should be willing to explain the benefits to the customer. Customers want products that work well and that are built on technology that is not at risk of obsolescence. Lisp is not in danger of obsolescence and offers significant productivity advantages. Hence, the case is not difficult to make, although management can be expected to require support from the technical staff in this regard.

In the past we have entered into contracts to extend Spike for other observatories. At times, some users have attempted the extension themselves with poor results. There is some feeling that our use of Lisp has been a contributing factor to the difficulty. If one is looking to engage a contractor on a short-term basis, it is certainly desirable to avail oneself of practitioners in commodity technology.

Lisp programmers are harder to find than Java programmers. Furthermore, for a customer wishing to engage in a minor, short-term effort, training is less likely to be an attractive investment. We are in sympathy with these views.

However, these views, while reasonable, neglect the more dominant factors. Our applications involve novel, non-trivial concepts and innovations in planning and scheduling. These are not easy to acquire of themselves. When coupled with the complex structure to be expected of any application of the scope of Spike, one finds a significant entry barrier which has nothing whatever to do with the implementation language. In order to credibly support third-party extensions to Spike, significant additional resources would have to be applied to the creation of design and API documentation and also to technical support. These activities are far beyond our present and expected future capacity.

We believe the current model of contracting to develop Spike extensions in house is the only feasible approach in the context of our current business model. If we were later to radically alter that model with a view toward supplying third-party customizable P&S products, it is far from clear that Lisp would not continue to be the ideal vehicle for the expression of the core services. Probably the natural course in that scenario would be deliver Spike as a Lisp-based DLL with an API layer that could be implemented in the language most convenient for the customer. There would be no need and no clear reason to constrain the API layer to be in the implementation language of the core DLL.

In summary, our position on customer attitudes is that while some customers can be expected to have reservations about Lisp, most of these reservations will be based on a lack of information or on misinformation. If we believe that we can deliver a high-quality product at lower cost by using Lisp, then management and the technical staff should partner to present a clear case for the customer's benefit.

10 Recommendation

While there is not universal consensus on all points, the tangible evidence known to the committe strongly supports the position that Lisp was and continues to be the most effective vehicle for planning and scheduling systems at the Institute. We find that while popular opinion is biased against Lisp, popular opinion is misinformed and in conflict with the concrete evidence.

The primary sources of controversy involve non-technical factors, particularly the risk of prohibitive staffing costs which might arise in the smaller Lisp market. We find this concern to be misguided given the nature of our application domain and our positive experience.

There *is* consensus that Lisp is the superior programming language, but lacks adequate multi-platform GUI support. Our recommendation is that we continue using Lisp for the domain modeling in conjunction with a Java-based, Swing front-end. This approach is common and has been demonstrated locally via the APT OrbitPlanner which uses the Lisp-based TRANS as the backend, with CORBA serving as an integration vehicle.

The remainder of the document presents reference materials including the aforementioned postion papers submitted by the members of the language committee. The position papers present the preceding arguments in more detail and many additional arguments to support our conclusions. Also included are testimonials from Paul Graham, an outside expert, who used Common Lisp to develop one of the first Web commerce applications (Viaweb) which was subsequently adopted by Yahoo.

11 Acknowledgements

The author thanks Mark Giuliano, Ron Henry, Ian Jordan, Bill Workman and the other members of the FPSG who, in addition to the members of the language subcommitte, made significant contributions to the form and substance of our report. Thanks also to Larry Hunter and Bruce Nilo for advice and conversation.

12 The Feature Matrix Exercise

Language Features

Feature	Value	Lisp	score	JDK 1.4+	score	Python	score		C++	score
access control	1	0	0	0.75	0.75	0	0		1	1
reflection	1	1	1	1	1	1	1		0	0
mop	1	1	1	0	0	0	0		0	0
rich dispatch	1	1	1	0	0	0	0		0	0
type composability	1	1	1	0.5	0.5	1	1		1	1
const variables	1	1	1	1	1	0	0	?	1	1
const methods	1	0	0	0	0	0	0		1	1
const method arguments	1	0	0	0	0	0	0		1	1
generics	1	1	1	1	1	1	1		1	1
dynamic typing	1	1	1	0	0	1	1		0	0
static typing	1	1	1	1	1	0	0		1	1
meta-programming	1	1	1	0	0	0	0		0	0
higher-order functions	1	1	1	0	0	1	1		0	0
lexical closures	1	1	1	0.5	0.5	0.75	0.75		0	0
multiple return values	1	1	1	0	0	0	0		0	0
functional	1	1	1	0	0	0	0		0	0
	14		11		5.75		5.75			7

Interoperability

Feature	Value	Lisp	score	JDK 1.4+	score	Python	score	C++	score
threads	1	0.75	0.75	1	1	0.5	0.5	1	1
portability	1	1	1	1	1	1	1	1	1
corba	1	1	1	1	1	0.5	0.5	1	1
sockets	1	1	1	1	1	1	1	1	1
language	1	1	1	1	1	1	1	1	1
speed performance	1	0.75	0.75	0.75	0.75	0.5	0.5	1	1
space performance	1	0.25	0.25	0.25	0.25	0.25	0.25	1	1
gui	1	0.5	0.5	1	1	0.5	0.5	1	1
	8		6.25		7		5.25		8

Non-technical

Feature	**Value**	**Lisp**	**score**	**JDK 1.4+**	**score**	**Python**	**score**	**C++**	**score**
3rd-party codebase	1	0.5	0.5	1	1	1	1	0.75	0.75
hiring pool size	1	0.25	0.25	1	1	0.5	0.5	1	1
commercial tools	1	0	0	1	1	0.25	0.25	1	1
maturity	1	1	1	0.25	0.25	0.25	0.25	1	1
standards (e.g. javadoc)	1	1	1	1	1	0.5	0.5	1	1
developer appeal	1	0.25	0.25	1	1	0.5	0.5	0.75	0.75
licensing cost	1	0.25	0.25	1	1	1	1	1	1
	7		**3.25**		**6.25**		**4**		**6.5**

References

[1] Erann Gat. Point of view: Lisp as an alternative to Java. *Intelligence*, 11(4):21–24, 2000.

[2] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York, NY, 1991. ISBN 0-07-032813-7.

[3] CORPORATE The Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal of Educational Resources in Computing (JERIC)*, 1(3es):1, 2001.

[4] Lutz Prechelt. Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, 1999.

The Value of Common Lisp

John Michael Adams

Truth scarce ever yet carried it by vote anywhere at its first appearance: new opinions are always suspected, and usually opposed, without any other reason but because they are not already common. But truth, like gold, is not the less so for being newly brought out of the mine. It is trial and examination must give it price, and not any antique fashion; and though it be not yet current by the public stamp, yet it may, for all that, be as old as nature, and is certainly not the less genuine.

—John Locke, An Essay Concerning Human Understanding

1 Introduction

Why do Lisp users believe it to be uniquely powerful and conducive to programmer productivity? If Lisp is really so good, why isn't it used more widely? If it is not widely used, doesn't that impose prohibitive staffing costs? The author addresses these questions, presenting evidence to support his claim that Common Lisp delivers unmatched power and productivity affording substantial cost savings on medium to large projects, even when allowing for programmers to learn the language on the job.

For those who have studied the matter for themselves, and acquired some experience with the language, Common Lisp is seen to be uniquely powerful and conducive to programmer productivity to a much higher degree than alternative languages. Those who do not know the language, perhaps only having encountered it briefly in a college survey course, or perhaps long ago, commonly suffer from debilitating misconceptions of Lisp. The author claims that the use of Lisp can significantly enhance the returns from our investments in software engineering labor. *To achieve our science goals at minimal cost, it is advisable that we understand the advantages of Lisp.* In support, the author presents evidence, both analytical and empirical, in addition to examples of remarkable commercial success, including the continued robustness of commercial Lisp vendors.

Languages obviously constrain the manner and content of our thought. Consequently, programmers tend to feel that whatever language they know is good enough. But, if one *is* interested in learning about a language, one cannot expect to go very far by listening to someone else talk about it, particularly if the concepts of the new language form a conceptual superset of those already known. Despite the difficulty, the author will later present a few such technical aspects of Common Lisp as he feels benefit himself and can perhaps inspire an interest in others.

In the end, the author can hope to do little more than comment on the color of the water, whereas to develop a credible understanding, the reader must dive in and swim for a while. Those who will not or cannot develop understanding for themselves must, of logical necessity, either accept the judgment of those who have, or else act on the basis of prejudicial sentiment. The author hopes that many will engage in earnest study of these matters, and stands ready to assist and encourage all who do.

Some are willing to accept the technical superiority of Common Lisp but hold that countervailing economic considerations dominate the technical ones so that, despite many fine qualities, Lisp does not turn out to be such a good investment. Others, who know Lisp well, believe the opposite, that Lisp is a superlatively *good* investment, that when seen clearly in perspective, fears of economic risk are found untenable. *Moreover, the author will show them to be in conflict with the empirical evidence.* Since, in the author's experience, this issue is at the heart of nearly all controversies concerning the use of Lisp, we begin by a consideration of economic matters.

2 The Economics of Lisp

2.1 Causes of Popularity

If Lisp is such a good thing, why is not more widely used? It is a good question and also easy to answer.

2.1.1 The Education Gap

The features of Lisp are not widely known. At some leading computer science schools such as MIT, lisp has a major role as the first language taught to undergraduates. At other schools, Lisp is not commonly taught outside of artificial intelligence courses. Often, Lisp is only briefly surveyed as part of the undergraduate programming languages course. The author had this experience in college and came away wondering why anyone would want to use such a language.[1]

Even recently published programming language textbooks are outdated in their coverage of Lisp. Some authors fall into the trap of attempting a shallow survey of many languages. This is problematic on two counts. First, it encourages premature judgment on the part of students. More importantly, languages evolve, and it is not practical to track the developments of many languages.[2] One textbook, used as recently as 1997 at Westchester and 1999 at Hamilton College, presents a horribly outdated treatment of Lisp sufficient to understand why no reader of the text could have the slightest appreciation for the actual qualities of the language. [14]. We will examine some of its material in order to see how university students are taught false things about Lisp. Note that the author's purpose is not to criticize the textbook nor its authors, but rather to show how what recently trained students have read about Lisp is false.

The author stresses that in order to make these points, he did not go searching for an extreme example. He found the book on his office shelf, his office mate and another colleague having used it in school. Below, the author calls out some of the defective content of [14] together with corrective commentary.

Falsehood 1: Datatypes in Lisp are rather restricted. Lisp has a full complement of datatypes including lists, trees, resizeable arrays, structures, classes, hash-tables, IEEE floats, strings, true rational numbers, symbols, and arbitrarily large integers. In addition, its facilities are well-suited to the design of any user-defined type whatsoever.

Falsehood 2: Lisp control structures are relatively simple. Lisp has all common control structures including the standard conditional and looping constructs and also exceptions. Moreover, unlike other languages, one can add new control structures via the macro facility.

Falsehood 3: Lisp does not distinguish between Lisp data and Lisp programs. At compile time, Lisp programs have a list representation which makes it easy for the programmer to introduce powerful code transforms (more on this later). However, the compilation process emits *machine language.*

Falsehood 4: Lisp has no real concept of statement execution. Those who are aware of the distinction between functional and imperative languages

[1]The author was mainly disturbed about having dynamic scoping by default. But this defect was long ago repaired. Common Lisp has lexical scoping by default.

[2]See [7] for a beautiful upcoming text which avoids this.

can guess at what the authors mean here. Without straying far afield, it suffices to observe that Lisp *does* have such a concept. Lisp "statements" are called expressions and these can be strung together, grouped into functions and loop bodies and so forth just like statements in any other language.

Falsehood 5: Lisp has always been a niche language and is not well-suited for most mainstream computer applications. The reader is invited to view the Franz, Inc. website which collects many examples of outstanding applications implemented in Common Lisp including applications in pharmaceutical research, computer gaming, airline fare search, B2B and others. These examples conclusively refute the notion that Lisp is a niche language.

University treatments of Lisp are egregiously outdated except in AI groups and islands of excellence such as MIT. Since many professors and textbook writers are out of date, so are the students who study with them. This accounts for the widespread ignorance of, and in some cases aversion to Lisp in the current software engineering community.

2.1.2 Forces that Oppose Learning

The easier or smaller or more ordinary a programming problem, the less does it matter which languages and tools are used. Most programming tasks do not present much intrinsic difficulty. [3] Even if the popular tools are not always the most cost-effective, people understand how to use them and can generally be successful because the intrinsic difficulty of the problem and the scale of the application do not suffice to make the weaknesses of ordinary tools a critical factor.

Moreover, if a better tool could have saved effort, this is not easily perceived. Not uncommonly, that the project was completed *at all* is considered cause for celebration. To make the differences among tools obvious, one would have to, in some appropriate sense, do the task twice, once with each tool—a universal impracticality for real projects.

Because programming labor is so expensive, most programmers are discouraged from investigating innovative tools when their time could instead be spent fixing the next defect. All of these circumstances combine with the result that programmers rarely have the opportunity to consider other than whatever tools are on hand. Thus, they have little basis for appreciating the value of unusually powerful tools in those scenarios to which the unique features of such most strongly apply.

[3]An intrinsic difficulty is one that arises from the nature of the problem as opposed to peculiar or unrelated characteristics of technology and tools

The majority programming languages are the languages optimized for the most common problems of the common programmer. Java is popular because it can readily address many common problems. *More people use Java for the same reason that more commercial airliners than F117A stealth fighters are in the air at any given moment.* [4] The former solves the more common problem, but the latter addresses requirements that are beyond the reach of the former.

To this we can add that, according to normal psychological mechanisms, programmers have a strong tendency to feel that whatever language they know is good enough for the task at hand. Paul Graham elaborates on this point in his eloquent testimonial, *Beating the Odds.*[5]

2.2 Staffing Costs

2.2.1 Perceived Risk

We can all agree that, other things being equal, a language nobody wants to use should be avoided if possible. If a language is not popular, it may be difficult to fill positions, or else one must invest in language training or even struggle to find willing learners.

The Planning and Scheduling Branch and its organizational antecedents have over a decade of experience in staffing Lisp projects. *Our direct experience contradicts the notion that it is difficult to attract excellent programmers to Lisp projects.* This experience agrees with the reports of others engaged in Lisp development with whom the author has corresponded.

But, it remains to consider the cost of teaching Lisp (or finding candidates that already know it), and the return that can be expected from the investment.

2.2.2 The Fundamental Logical Error

Some argue that because there are fewer Lisp programmers than Java programmers, those who adopt a Lisp strategy can expect to incur costs associated with hiring and training. The logical error comes in trying to deduce from this agreed upon proposition, that Lisp is a bad investment. Such an argument, while raising a valid concern, is ultimately erroneous, even nefarious because it considers *only* cost and fails to account for the *benefit* that might be gained from the investment. Programming languages vary in power.

[4]According to CNN, approximately 40,000 commercial aircraft are airborne over the U.S. alone at any given moment.

[5]The complete text is included in the appendix.

Admittedly, it is not so easy to account for these kinds of returns because some of the contributing elements are subtle and are difficult to perceive by casual glance or in a single place or over short time periods. Nevertheless, if we are to avoid basing our action on ignorance and prejudice, we should make some effort to understand what these returns might be. Fortunately, there is evidence that Lisp reduces cost and risk in software development.

2.2.3 The Work of Prechelt and Gat

In an important study [12], Lutz Prechelt compared, among other language related factors, the productivity of a sample of programmers using C/C++ versus Java on the same programming task. Eran Gat of JPL extended this study to include Lisp [4].

Gat found that,

> Lisp's performance is comparable to or better than C++ in execution speed; it also has significantly lower variability, which translates into reduced project risk. Furthermore, development time is significantly lower and less variable than either C++ or Java...
>
> Programmer experience cannot account for the difference. The experience level was lower for Lisp programmers than for both the other groups (an average of 6.2 years for Lisp versus 9.6 for C and C++ and 7.7 for Java). The Lisp programs were also significantly shorter than the C, C++, and Java programs.

2.2.4 The Work of Capers Jones

In Applied Software Measurement [8], Capers Jones, the noted software engineering and measurement expert, quantifies the unit costs of writing software by estimating number of source code statements required per function point for many programming languages. [6]

The number given for Common Lisp is 21; for Java, 53. In other words, it is estimated that for a given level of functionality, a programmer writing in Java will need to produce twice as much code as a programmer coding in Lisp. Other things being equal, this translates directly to higher development and maintenance costs.

[6]This table appears on page 76 of the first edition. Published in 1991, this edition does not include relevant data on Java and Common Lisp. An updated table including entries for Java and Lisp (the relevant entry is actually CLOS, which includes the object-oriented facilities of Common Lisp) can be found at http://www.theadvisors.com/langcomparison.htm .

The work of Gat and Capers Jones provide clear, independently confirmed evidence in favor of Lisp.

2.2.5 Costs of Learning

There is also evidence that Lisp is easy to learn and that once learned helps people produce better software with less code and less effort. I will mention one example here.

Here is one university professor's account comparing the use of Scheme [7] and Java in introductory programming courses.

> While OOP may not be too "advanced" for beginning programmers, Java may be, because it has so many syntax rules and requires so much overhead code for a typical CS1 program of a few dozen to a few hundred lines. ... Qualitative results, from the instructor's perspective are clear: the CS1 class in Java was largely about syntax and platform, while the CS1 class in Scheme was largely about data structures, function composition, and software engineering. It felt like teaching. [1]

As evidenced by this account, Lisp is easy to learn. Space Telescope has over a decade of experience in training Lisp programmers. We have found it to be decidedly unproblematic because Lisp is so easy to learn, but even more so because the effort required to learn any programming language is a small fraction of the effort required to understand a complex application domain such as ours.

2.2.6 Putting Language Learning Costs in Perspective

The absolute cost of language training is, by itself, of little interest. It is only when this cost is compared to the total expected training cost, including factors unrelated to languages, that we understand the impact of language learning costs and can compare these with the benefits in order to render credible judgment.

There are varying degrees of fluency in programming languages, just as with natural language. At the beginning level one is able to give expression to simple and ordinary ideas. In programming, this means (roughly) having a mastery of control structures, basic I/O facilities and the elements of encapsulation. At

[7] the other major Lisp dialect apart from the ANSI Standard Common Lisp used at the Institute

this level one can code to specification or within a given framework. With more or less direction according to experience and circumstance, one can accomplish useful tasks of small to medium scope, perhaps working at the function or module level. A competent professional can attain this degree of fluency in any high-level language in a matter of days. The cost of acquiring this degree of fluency is negligible. [8]

At the intermediate degree of fluency, one can handle tasks of intermediate to large scope including non-trivial designs. If the language involves concepts new to the programmer, then attaining this degree takes weeks or months according to circumstances. These costs may or may not be negligible depending on the duration and complexity of the project. For longer and more complex projects, this cost *is* likely to be negligible because the cost of acquiring domain and application specific knowledge begins to overwhelm the cost of acquiring language knowledge. [9] The great majority of programmers never advance beyond the intermediate level. Consequently, consideration of higher levels of expertise can be neglected in the aggregate analysis.

For shorter projects or projects of a simple nature, it is wise to avoid this type of training cost as much as possible. But planning and scheduling applications in general, and those associated with Space Telescope in particular, are not of this variety.

Thus, the author concludes that the cost of on the job language learning is only significant for projects that are short or simple or necessarily deal mostly with short-term labor. Space Telescope P&S projects have never been of this variety. Hence, our projects are ideally suited to enjoy the economic advantages of a Lisp strategy.

Again, the evidence supports the conclusion. In stark contrast to common conception, Carl de Marcken of ITA software remarked:

> We've had very little trouble getting non-Lisp programmers to read and understand and extend our Lisp code.[6]

This is not the testament of an academic writing toy applications. This is the testament of someone who has used Lisp to displace major corporations in the competitive fare search market.

[8]Consequently, when there is no shortage of tasks of small to medium scope, those in hiring roles are wise to hire (permanent employees) on the basis of insightfulness, maturity and creativity rather than so many years in such and such a technology. In disregard of this principle, many err by attempting to optimize a staffing variable of minimal impact.

[9]In these circumstances, the required domain and application knowledge not only is orders of magnitude more complex, it is also orders of magnitude more difficult to acquire since the majority of knowledge tends to be distributed in the minds of many people rather than collected into readily available sources such as books and training courses.

2.2.7 The API Space

It is important to consider how the availability and scope of both intrinsic and third-party APIs bears on development costs. It is of concern if a significant portion of an application can be coded immediately on top of available APIs in one language, whereas in another all of this must be written from scratch.

The author again claims that this is negligible in our case. First, we are able to invoke Java directly from Common Lisp using a technology called JLinker. Moreover, through the Lisp foreign function interface we can also call Fortran and C/C++ functions. Thus, choosing Lisp in no way divorces us from the Java codebase nor any other codebase.

Finally, the corpus of Java code is less applicable to our applications than to the average application. A COTS subcommittee, a sibling of the language group whose work we document here, found that COTS products did not exist that were likely to meet our needs in a cost-effective way. Our development efforts are dominated by the development of non-commodity algorithms which are not supported in the Java corpus.

2.3 The Lisp Market

This section explores some characteristics of the Lisp market including the universe of available Lisp implementations, the state of the commercial market, the financial condition of our particular vendor and some examples of remarkable commercial success.

2.3.1 Lisp Implementations

There are many high quality Lisp implementations including several actively maintained free ones. Despite popular misconception, the demand for Lisp systems is fairly robust, having supported two commercial vendors for many year, Franz, Inc. and Xanalys (formerly Harlequin).

These companies are privately held and so financial information is not publicly available. According to our Franz sales rep, Franz is profitable and has been for many years.[10] Yet, perhaps the best way to evaluate the condition of a technology vendor is to observe their customers.

[10] Franz began doing business in 1984 and has remained strong, having weathered the famed AI winter.

Commercial Lisp companies have many Fortune 500, utility and government customers. There is no shortage of examples of famous and successful companies using their products successfully. One notable contemporary example is ITA software which supplies the fare search backend to Orbitz. The nature of pricing in the commercial airline industry involves a combinatorial explosion that must be dealt with by fare search programs.

The ITA accomplishment demonstrates that Lisp scales very well and can handle problems with extreme performance constraints. The ITA software runs under both Allegro Common Lisp and CMUCL which is a high performance free Lisp implementation originally developed at Carnegie Mellon University with DARPA funding. To see more detailed comments on the industrial use of Lisp, the reader is directed to comments of Carl de Marcken[6].

Another famous Lisp application is the popular Jak and Dexter game for Sony Playstation. Allegro Common Lisp [11] was used to write the game engine. Consult the Franz website to see many more examples of recent commercial lisp applications.

2.3.2 Open Source Lisp Implementations

There are many free Lisp implementations.[12] Two in particular stand out as being feasible targets for our applications in the future The first is CMUCL, mentioned earlier. CMUCL, like the commercial implementations, compiles lisp to native machine language. It is a high quality implementation with sophisticated type-inference and type-checking features. A number of industrial strength applications use CMUCL, including Orbitz mentioned earlier.

The other prominent free Lisp is CLISP. CLISP is a high quality implementation but is interpreted. Viaweb (now called Yahoo Stores), is one example of a famous application implemented in CLISP. For more detail, see the articles of Paul Graham[13].

Support for CLOS [14], varies more than desirable among free implementations. Dealing with this issue would be the principal cost of porting to a free lisp. PSDB[15] applications, and particularly TRANS[16], make use of the most advanced features of CLOS. A certain amount of research and development would

[11]This version of Lisp used at the Institute.

[12]While the author recommends using one of the commercial implementations, he was asked specifically to discuss open source alternatives.

[13]www.paulgraham.com

[14]Common Lisp Object System, the layer of Lisp that supports object oriented idioms.

[15]Planning Systems Development Branch

[16]TRANS is a Lisp application used in planning the detailed structure of Hubble observing activities given a high-level specification from an astronomer

be needed to cope with the weaker CLOS support of free Lisps. Thus, the cost of moving to a free Lisp would certainly exceed the cost of using one of the commercial implementations. The author recommends that if we use Lisp, we should only use the commercial implementations for the bulk of our work unless CLOS were deemed to be non-critical. [17]

2.4 Enabling Breakthrough Science

Historically, people have not coped well with software problems. In this light, if we hope to perform high service to our institutions, and in particular, if we are so ambitious as to aspire to *Enabling Breakthrough Science*, we can expect a difficult path requiring unusual insight, something not often found to coincide with popular opinion. Therefore, we should be skeptical of forming a technical strategy guided by fear of being unlike the majority. To do so is to practically guarantee an average or below average return on our investment in software engineering labor.

Instead of *following* the market, we should think in terms of *outperforming* the market. Common Lisp will help us do that in the area of software engineering, as it has demonstrably helped others, and it can help us do that not only in the area of planning and scheduling but in other areas as well.

3 The Technology of Lisp

With a few very basic principles at its foundation, it has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of, in a sense, our most sophisticated computer applications. LISP has jokingly been described as "the most intelligent way to misuse a computer." I think that description a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts [3].

–Edgar Dijkstra [18]

[17]Unlikely in our environment, but this has been the case for two important applications of which the author is aware: Doug Lenat's Cyc and ITA Software's fare search software which has such stringent performance constraints as to preclude OO dispatching.

[18]Noted computer scientist and recipient in 1972, of the Association for Computing Machinery's Turing award, often regarded as the Nobel prize for computing.

3.1 The Epsilon Effect

In the remainder, the author will try to explain the value seen in Lisp by its experienced programmers, many of whom have come to Lisp after much experience with other more popular languages.

In the eyes of the author, the power of lisp emerges synergistically from many features. Some features are of a seemingly small nature, but combine into something larger. In this connection, the author, trained as a mathematician, cannot resist the temptation to recall the harmonic series to the readers attention.

$$\sum_{n=1}^{\infty} \frac{1}{n} = \infty$$

When discussing the smaller features of technologies and of programming languages in particular, one can almost always make a fair argument, given any specific feature, that it is not too painful to do without. Such arguments are not wrong. But our regard for them should be moderated by recalling that small things do add up, as we are informed by mathematical fact.

This principle finds expression elsewhere.

> Hold not a deed of little worth, thinking ‘this is little to me’. The falling of drops of water will in time fill a water-jar. Even so the wise man becomes full of good, although he gather it little by little.
>
> Hold not a sin of little worth, thinking ‘this is little to me’. The falling of drops of water will in time fill a water-jar. Even so the foolish man becomes full of evil, although he gather it little by little.
>
> –The Dhammapada[19]

The point of all this is that small things can add up and interact in ways that are not so easy to see. The reader is encouraged to bear this in mind for the remainder.

3.2 Macros

> The Common Lisp macro facility allows the user to define arbitrary functions that convert certain Lisp forms into different forms before evaluating or compiling them. This is done at the expression level, not at the

[19] A profound and beautiful Buddhist text in the Pali language.

character-string level as in most other languages. Macros are important in the writing of good code: they make it possible to write code that is clear and elegant at the user level but that is converted to a more complex or more efficient internal form for execution [9].

–Guy Steele

The most unique and important feature of Lisp is its macro facility. Lisp macros have little in common with macro facilities of the same name in other languages. Macros are program writing programs that operate at compile time, after code is parsed but before it is compiled. Macros are intended for the user to construct new programming idioms to suit the application at hand.

The practical possibility of such a feature derives from Lisp's syntactic regularity and extreme simplicity. At the lexical level, everything is a list. The macro itself is a function that, roughly speaking, receives unevaluated expressions from the parser and then returns something else that is to be considered the result of that parsing. This something can then be compiled or evaluated. This wouldn't work in Java because there are so many syntactic entities and relationships.

The macro facility allows one to introduce new idioms in a transparent way. Applications can range from simple convenience idioms, to new control structures, to entirely new programming paradigms. Macros make Common Lisp uniquely extensible among programming languages.

As a simple example, consider the standard lisp macro `with-open-file`, an example of a context providing macro. It's purpose is to setup an input/output context in which to execute user specified code. The context guarantees that the file will be properly closed when control leaves the with-open-file body, regardless of whether of whether control leaves by falling out of the user body or because of a system or user thrown exception.

The code below shows the definition, example use and also the macro expansion. The macro expansion is the final form of the code prior to compilation. This macro is simplistic and is intended to give the reader a sense of macro mechanics.

```
;;; Defining the macro

(defmacro with-open-file ((stream &rest open-args) &rest body)
  (let ((x (gensym)))
    `(let ((,stream (open ,@open-args))
   (,x t))
```

```
       (unwind-protect
   (multiple-value-prog1 (progn ,@body (setq ,x nil))
     (when (streamp ,stream)
       (close ,stream :abort ,x)))))))

;;; Using the macro

(with-open-file (s "dump" :direction :output)
  (princ 99 s))  ;; print 99 to the stream s

;;; Macro expansion.  This is what the compiler sees.

(LET ((S (OPEN "dump" :DIRECTION :OUTPUT)) (#:G1000 T))
     (UNWIND-PROTECT
      (MULTIPLE-VALUE-PROG1
       (PROGN (PRINC 99S)
      (SETQ #:G1000 NIL))
       (WHEN (STREAMP S) (CLOSE S :ABORT #:G1000)))))
```

Perhaps C++ templates are the closest thing to Lisp macros in common use. But C++ macros are just type parameterized boilerplate generators hard-wired to do one thing. [20] Lisp macros are infinitely more powerful because one can write arbitrary Lisp to define the macro expansion. That is why it is true to say that macros are Lisp programs that write other Lisp programs.

The beauty and power of Lisp macros is that they permit users to craft language extensions which are indistinguishable from system features. In fact, many features of Lisp are implemented at the user level by macros. One can write new macros around these to customize the system behavior. Of course, it is foolish to do such things needlessly. Often, however, macros allow one to create on top of Lisp, a new language that more naturally expresses some desired computations in a special domain. This allows one to write cleaner, more elegant programs.

To get a feel for the power of macros, consider that CLOS, the Common Lisp Object System, was written on top of Lisp. The basic building blocks of Lisp, while simple, are so fundamental and powerful, that entirely new programming paradigms can be built upon them transparently. With other languages, one gets just what the language designers give you and nothing more. The language cannot be tailored in a natural way to the application. To get any new innovative language features, one must wait for a new compiler to be written.

[20]The author would argue that templates are more of a workaround for static typing than a primary feature.

Macros are not an elementary subject. Paul Graham has written an entire book about them[5]. We have given a simplistic example here. Unfortunately, the more impressive the example, the less readily can it be made clear to the casual reader in a small space. However, see Section 3.7 for a mostly narrative treatment of a real world example involving macros and some other things.

3.3 The Common Lisp Object System

3.3.1 The Question of Multiple-Inheritance

There are two kinds of multiple inheritance which are commonly discussed: multiple inheritance of classes, which is to say of behavior and data structure, and multiple inheritance of interfaces, which is to say, of functional interface. In what follows, the unqualified use of the term *multiple-inheritance* (MI) refers to multiple-inheritance of *classes*.

There has been much discussion over the years (before and after the advent of Java) concerning the relative importance of these two varieties of inheritance. [21]

Lisp, Python and C++ support multiple MI. Java supports only MI of interfaces. Is this a serious weakness of Java or is it a shrewd pruning of a non-essential feature? Perhaps it is both.

Recall that Oak, the predecessor to Java, was originally designed as an embedded language for appliances. In this context, it makes sense to accept some loss of expressiveness in exchange for simplicity of implementation. The constraints of embedded environments normally neither accommodate nor require large and complex class hierarchies.

From purely analytic considerations, lack of MI can be expected to cause trouble because it imposes synthetic asymmetry on designs and a consequent asymmetry and verbosity on their implementation.[22] [23] The difficulty arises when it is desirable to inherit behavior [24] from multiple classes. Here Java programmers must resort to composition and forwarding, by which is meant that, instead of inheriting from a second class, one must create an instance of it, and for all of its methods, implementations which do nothing more than call the implementations of this "artificial base class".

[21]See [13] for a 1993 panel discussion.

[22]Asymmetric in the sense that rather than inheriting from two classes, one must treat one via inheritance and the other via composition.

[23]Recall the Capers Jones estimate that Java applications will on average contain twice as many lines of code per function point.

[24]The argument applies similarly to data structure.

This coerced forwarding is unfortunate because it violates parsimony, forcing the programmer to include many lines of code which have nothing to do with the application. Perhaps more importantly, because of the synthetic asymmetry, it violates the principle that software models should correspond in a natural way to the real world objects which they represent.

Some argue that inheritance can break encapsulation in a bad way. By this they mean that super and subclasses may have knowledge about the internals of one another such that it becomes difficult to extend either.[25] In the author's view this observation says nothing but that good design requires discretion.

The fact of the matter is that many problems can be elegantly cast in terms of MI and that to exclude this feature from a language is to exclude these programs and replace them with inelegant verbosity.

Some others argue that it is easy for programmers to make a mess of things with MI and that therefore the feature is bad, notwithstanding many valid use cases. The present author does not subscribe to this sort of philosophy, believing instead that programmers should be given the most powerful language available and taught to use that power wisely.

Common Lisp does have multiple-inheritance. We have made frequent and successful use of it in complex applications, both for subclassing and also in the mixin style [2].

The author believe that the designers of Java erred in choosing not to support multiple inheritance of behavior and data structure, not, perhaps, with respect to its originally intended application space, but certainly with respect to complex applications solving hard problems.

3.3.2 The CLOS Generic Dispatch

Orthogonal to the matter of MI, we now consider method dispatch. Method dispatch refers to act of choosing, by consideration of types of one or more arguments, which method implementation should be called from among many possibilities. Java, C++, and Python all use what is called single-dispatch. This means that the implementation is selected on the basis on one argument (the first). Because of the distinguished role of the first argument, it is omitted from the method's argument list as specified by the programmer. With single-dispatching, when an operation's meaning depends on the type of multiple arguments, one is forced to refer explicitly to the type of the second, and any subsequent arguments in order to manually complete the dispatching (as it

[25]There are other points of debate having to do with resolving diamond inheritance and also the impact of linearizing base classes, but treating these subjects would require undue.

were).

CLOS dispatches methods on the basis of the types of all arguments. For example, one might want event handling to depend on the class of the recipient and also the class of the event. With single-dispatch, if distinguishing recipient classes were desirable, one would probably dispatch on the recipient and let it select its action based on the kind of event. Multiple-dispatch gives a more natural expression to this allowing the explicit selection to be omitted. Thus, CLOS dispatching is considerably more general than in other language, affording increased expressive power.

3.3.3 Method Combinations

Relative to other OO implementations, perhaps the most unique feature of CLOS is method combinations. In other languages, a call to a method is resolved into a unique call target corresponding to the most specific type of the distinguished first argument. That's it.

In contrast, CLOS defines a rich dispatching framework that allows multiple functions to be called and their arguments to be combined in configurable ways as the value of the generic (virtual) function call.

In a paper describing Flavors, an ancestor of CLOS, David Moon summarizes examples of how method combinations can be applied[11].

> Some example of built-in method-combination types are:
>
> - Call only the most specific method
> - Call all the methods, most-specific first or in the reverse order
> - Try each method, starting with the most specific, until one is found that returns a value other than nil.
> - Collect the values of the methods into a list.
> - Compute the arithmetic sum of the values of the methods
> - Divide the methods into three categories: primary methods, before-daemons, and after-daemons. Call all the before-daemons, then call the most specific primary method, then call all the after-daemons.
> - Use the second argument to the generic function to select on of the methods.

In certain applications this affords considerable reduction in complexity and improved modularity. This is possible because the implementation of an operation can be distributed among several classes with each implementing the portion that pertains to their own type and without the need to write any application code to explicitly coordinate their execution and combination.

Moon explains.

> When defining a method, the programmer only thinks about what that method must do itself, and not about details of its interaction with other methods that aren't part of a defined interface. When specifying a method-combination type, the programmer only thinks about how the methods will interact, and not about the internal details of each method, nor about how the method-combination type is implemented. Programming an individual method and programming the combination of methods are two different levels of abstraction. Keeping them separate promotes modularity[11].

Corresponding to the sixth item in Moon's list, CLOS defines what it calls the *standard method combination* which suffices for most purposes. In addition, there are several other simple built-in combination types including: list, and, or, max, min and some others. These combine values in the natural way.

CLOS also permits the user to define arbitrary method combination schemes. We will not cover user-defined combinations here except to remark that they can have a substantially positive affect on robustness and modularity by allowing the programmer to avoid webs of explicit forwarding and superclass method invocations.

In the standard method combination, also called the *generic dispatch*, there are four varieties of methods which are specified by qualifiers in the method definition. These are :around, :after, :before and primary. Primary methods are specified by omitting any qualifier. Only primary methods and :around methods can affect the value of the generic function. :Around, :before, and :after method perform auxiliary or administrative tasks such as resource management (e.g. acquiring and releasing a semaphore or connecting to a database), result validation, and debugging.

Figure 1, reproduced from [10], illustrates the control flow associated with the standard method combination.

After Method Example

The author once implemented a facility to record CORBA method invocations both for tracing purposes regression testing. In order to do this, the author broke into the debugger in a CORBA method invocation, observed the vendor methods in effect and wrote an :after method on the dispatcher to receive the operation and arguments to be recorded.

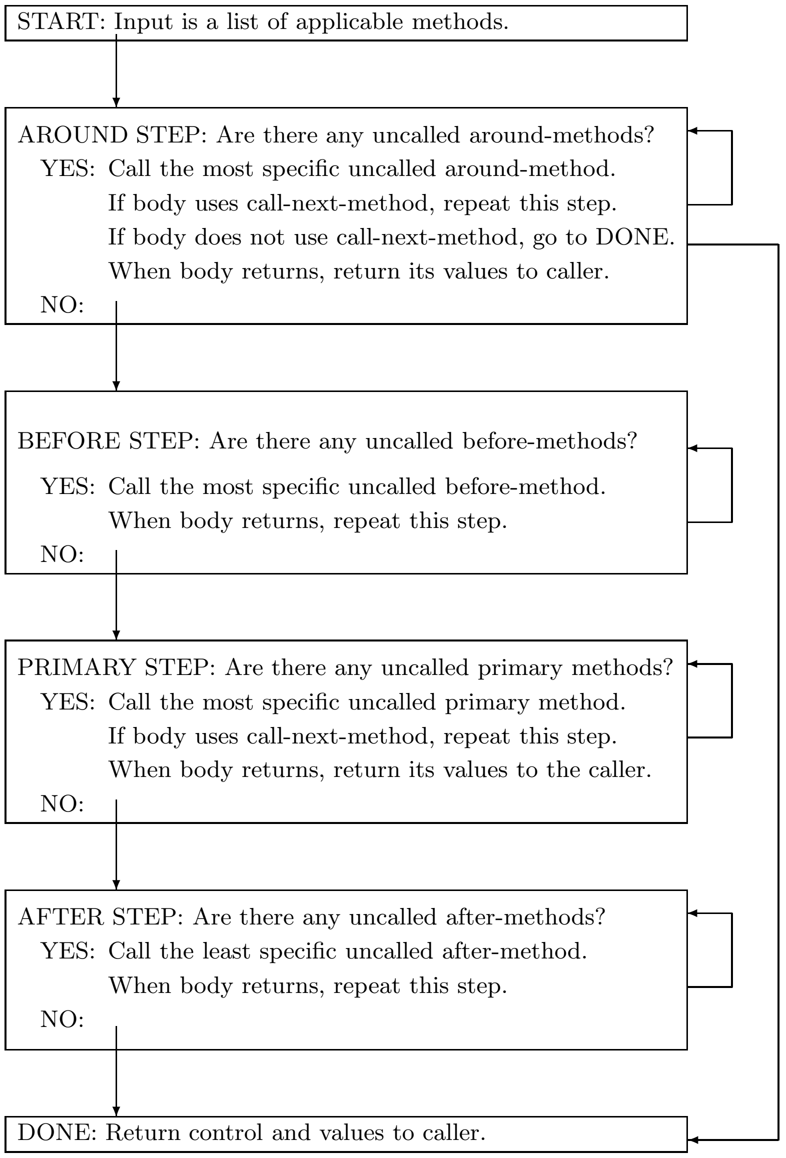
START: Input is a list of applicable methods.
AROUND STEP: Are there any uncalled around-methods?
YES: Call the most specific uncalled around-method.
If body uses call-next-method, repeat this step.
If body does not use call-next-method, go to DONE.
When body returns, return its values to caller.
NO:
BEFORE STEP: Are there any uncalled before-methods?
YES: Call the most specific uncalled before-method.
When body returns, repeat this step.
NO:
PRIMARY STEP: Are there any uncalled primary methods?
YES: Call the most specific uncalled primary method.
If body uses call-next-method, repeat this step.
When body returns, return its values to the caller.
NO:
AFTER STEP: Are there any uncalled after-methods?
YES: Call the least specific uncalled after-method.
When body returns, repeat this step.
NO:
DONE: Return control and values to caller.

Figure 1: Flow of control of standard method combination

Several points are in order. First, the vendor did not provide a tracing implementation that would support CORBA method recording and playback. Second, the author confirmed with the vendor that his method was a reasonable and safe workaround until such time as they provided a canonical interface. Third, the author was able to solve the recording problem immediately, without having to wait for a vendor patch or release which, even then would necessitate regression testing before delivery.

By leveraging the standard dispatch, we were able to participate in the vendor protocol without any prior design on the vendor's part. Thus, we were able to implement an extraordinarily valuable feature in a timely manner. If we had been using Java, we would have been out of luck, plain and simple, until the vendor provided a solution.

Note that we did not need the vendor sources in order to write the after method. In CLOS methods do not belong to classes; they belong to generic functions. When calling a generic function, the system determines which methods to call by inspecting the methods associated with the generic function in order to find the ones applicable to the supplied arguments. Because methods are associated with generic functions, one does not need lexical access to the class definition. In Java, methods must appear in the lexical scope of their class and therefore source would be required even if :after methods were present in the language.[26]

By doing something new in the after method, we are able to incrementally alter behavior in a benign way. The :after method body is isolated in its own method definition and so there is no danger of inadvertently changing the existing code. Furthermore, since the :after does not contribute a computational result in protocol, again there is no possibility of inadvertently altering the method's return value.

3.4 Multiple Return Values

Lisp functions may return multiple values. However, the caller can ignore any or all of them. If the caller does not care about the additional return values, he can treat the function as if it returns a single value. If `myfun` is a function returning say, three values, one can treat these as follows.

```
;; Print the first return ignoring others
```

[26]There is an experimental version of Java from Xerox PARC, called AspectJ which offers some support in this regard. But notice how in order to get the new power, somebody had to write a new compiler. With macros and other Lisp powers, one need not wait for wizards to deliver language feature. The author recommends AspectJ to Java programmers if only as an enhanced debugging and development facility.

```
(print (myfun))

;; Print the set of returns as a list
(print (multiple-value-list (myfun)))

;; Refer to the first two as r1 and r2
;; passing them to foo.
(multiple-value-bind (r1 r2) (myfun)
  (foo r1 r2))
```

Sometimes it is natural to use multiple values when a function computes several interesting things in natural course. For example, the truncate function returns a quotient and a remainder.

Multiple values can also be used to extending existing functions without affecting existing callers. Suppose `myfun` was originally written to return α. Further suppose that subsequently it were desirable to also return the intermediate result β. In other languages, this presents a problem since one would normally have to change all of the callers of `myfun` in concert with the change to `myfun` itself, for example by changing an int return type to a structure of two ints.

In Lisp, this is not a concern. Existing callers that access only the first return value, will still, following the `myfun` change, see only one return value. No recompilation will be necessary. On the other hand, new callers can be written to take advantage of the second return value. This is one example of how Lisp programs can be incrementally changed with *considerably less risk* than other languages. In another language, one would have to find and change all existing callers. Besides being tedious and error prone, it might also be practically impossible if some of the existing callers are in externally distributed libraries. This is but one of many ways in which Lisp permits programs to be evolved quickly, elegantly and with considerably less risk than other languages.

3.5 Variable Length Arguments

Similar comments hold for arguments to functions and methods in Lisp. Once can add new optional parameters to a function without needing to recompile existing callers. Java does not have variable length argument lists. This is one way in which Java makes it more difficult to evolve programs.

3.6 Dynamic Typing

Finally, Lisp has dynamic as opposed to static typing, meaning that objects have type rather than variables. Statically typed languages arise from the notion that type errors form a large class of runtime errors and should be eliminated so far as possible by compile time checking of calls against interfaces. The disadvantage of static typing is that it precludes many correct and useful programs. Moreover, heterogeneous collections are not strictly possible.

Dynamically typed languages arise from the notion that having to specify types for every variable needlessly slows initial development. Moreover, the majority of expressions do not have type errors. So why throw out all those useful programs?

It is worth noting that dynamic and static typing are not at all mutually exclusive inasmuch as one can augment any statically typed language with a new tagged-data type[7].

The crucial point is that while Lisp better supports rapid prototyping via its flexible typing system, it permits compiler type-checking and optimizations via optional type declarations. Thus, if a programmer or project finds static typing indispensable, a policy of type declarations can be implemented.

In the author's experience, range errors such as divide by zero or null pointer references, against which statically typed languages do *not* protect, greatly outnumber type errors at runtime. Thus, forcing programmers to write type specifications for every variable is misguided and increases development costs.

3.7 An Advanced Example

In this final example, the author will try to give a sense of how the semantics of Lisp can be extended in application specific ways to make difficult problems manageable. This example comes from the TRANS system, a planning system which embodies complex models of instrument capabilities and operations policies to transform specifications for astronomical observations to a concrete activity plan for the Hubble Space Telescope.

The TRANS system implements instrument activity models in which many complex relationships obtain between model variables. In earlier years, it was difficult to ensure that changes to one model variable were correctly propagated to other areas of the model. This led to bugs when the relationships were misunderstood, and to performance problems when calculations were repeated "just in case" when the relationships proved too difficult to analyze.

The TRANS team solved this problem by leveraging the power of Lisp macros and the power of the CLOS metaobject protocol. This solution is called the Constraint Sequencing Infrastructure (COSI). It is implemented in two major parts.

First, we introduce a macro, `defconstraint`, which is a wrapper around the standard system macro, `defmethod`, used to define methods. The application programmer uses `defconstraint` in exactly the same manner as `defmethod`. Under the covers, however, a special invocation context is setup around the body of the method which allows the system to know which constraint is running at a given time and to handle exceptions in a standard way.

Using the metaboject protocol, we are able to hook into the operation of reading and writing class fields (called slots in CLOS parlance). Because of the invocation context established with the `defconstraint` macro, the slot accesses can be associated with a particular method invocation. Thus, the system is able to understand which computations depend on which slots of which classes. When the value of a class slot is changed, the system reinvokes all methods that read that slot in the course of its computation.[27]

3.7.1 A Demonstration of COSI

The following example demonstrates, from perspective of the application programmer, how COSI can automatically manage relationships between the fields of objects. Unavoidably, in order to present the essence of the mechanism in a small space, we must scale the example below the point at which sophisticated techniques can be justified. The reader should ignore the application semantics of the example and should instead pay close attention to how the number-c slot is treated.

First, let us define a class, foo, with three slots: number-a, number-b and number-c.

```
(defclass foo ()
  ((number-a :initarg :number-a :accessor number-a)
   (number-b :initarg :number-b :accessor number-b)
   (number-c :initarg :number-c :accessor number-c))
  (:metaclass constraint-sequencer))
```

This is a standard CLOS defclass form. It uses the `:metaclass` option to specify a custom metaclass. The metaclass provides a way of associating

[27]The actual invocations are triggered by a call to `propagate`.

metadata with instances and serves as the focal point for hooking the slot reader and writer methods within the metaobject protocol. We will not include these methods because they do not concern application programmers, but the reader should understand that, as part of COSI, we have written code that operates behind the scenes, getting control whenever a slot is read or written.

Here we write the method that implements the relationship between the three slots of foo. We also define a constructor.

```
(defconstraint assign-number-c ((a foo))
  "Defines the rule for populating number-c"
  (setf (number-c a) (+ (number-a a) (number-b a))))

(defmethod initialize-instance :after ((a foo) &key)
  "Attaches the assign-number-c constraint"
  (assign-number-c a))
```

The constraint is called explicitly inside the constructor to establish this link. Thereafter, it is only called by the propagator. Typically, number-a and number-b will not have been populated at first. The Lisp system will signal a slot-unbound exception whenever a slot has not been initialized. If you examine the definition of `defconstraint` in the appendix, you can see that the macro inserts special exception handling code around the user's method body. In COSI, slot-unbound exceptions are ignored inside constraints under the assumption that the values will be provided later.

In order to ensure that the model is consistent prior to reading, constrained objects export their external slots via `defquery` forms. `defquery` is another COSI system macro that wraps the user's method body around another form that calls `cosi:propagate` before executing the user code.

```
(defquery get-number-c ((a foo))
  "Runs cosi:propagate before calling the number-c accessor"
  (number-c a))
```

Here we show how to exercise the simple model inside the Lisp interpreter. Throughout, we display the state of the foo object using the system provided function describe, First we create an instance of foo. The initialize-instance method runs implicitly as part of this.

```
Transformation(8): (setq f (make-instance 'foo))
#<FOO @ #x63eac42>
```

All of the slots are initially unbound.

```
Transformation(9): (describe f)
#<FOO @ #x63eac42> is an instance of #<CONSTRAINT-SEQUENCER FOO>:
  NUMBER-A              <unbound>
  NUMBER-B              <unbound>
  NUMBER-C              <unbound>
```

Now, let's give values to number-a and number-b.

```
Transformation(10): (setf (number-a f) 3 (number-b f) 4)
4
Transformation(11): (describe f)
#<FOO @ #x63eac42> is an instance of #<CONSTRAINT-SEQUENCER FOO>:
  NUMBER-A              3
  NUMBER-B              4
  NUMBER-C              <unbound>
```

Now call the `number-c` accessor via the `defquery` form defined earlier. The `assign-number-c` constraint runs implicitly as part of this.

```
Transformation(12): (get-number-c f)
7
```

Now we see that the new values for `number-a` and `number-b` have been propagated to give the new value of `number-c`.

```
Transformation(13): (describe f)
#<FOO @ #x63eac42> is an instance of #<CONSTRAINT-SEQUENCER FOO>:
  NUMBER-A              3
  NUMBER-B              4
  NUMBER-C              7
```

3.7.2 Remarks on the Example

Common Lisp is designed as a programmable programming language capable of supporting as yet unimagined solution paradigms. To implement a facility such as COSI in another language, such as Java, one would be forced to use either intrusive and error-prone coding conventions, or else one would have to resort to an external code generating program. This would be problematic due to build complications and the need to distinguish between generated and hand-written code. This is the Achilles heel of these other languages, that their semantics is statically defined. The impact of this weakness is negligible for simple, common problems, but becomes significant as the problem domain increases in complexity. Because the semantics of Lisp can be extended, Lisp is supreme in its ability to attack the most difficult programming problems.

4 Conclusion

The author has tried to shed some light on the nature of Lisp, particularly with regard to attitudes and economics. The author hopes the reader will come away with some sense of the uniqueness and power of Lisp as a classic tool of software engineering with long-standing and continued profound relevance to the difficult software problems of our time.

If we aspire to uncommon accomplishments, then we had better look for uncommon means. To follow the mass of popular opinion because it seems safe invites mediocrity and higher costs.

λ

A COSI System Macros

```
(defmacro defconstraint (name parameter &body body)
  "This macro is used to define a constraint. A constraint is a special
of method that has additional semantics associated with it.

Specifically:
  * A constraint indicates a piece of code that may be tagged
    as dependent on specific slot values. When those slot values change
    the constraint will be re-run.
  * A constraint is applied to a single object type. It would normally
    enforce the contained code for an instance of that type.
  * A constraint has no useful return value. This prevents unintended and
    unnoticed dependencies from being created through code which depends
    on the return value."

  (multiple-value-bind (dec doc bod) (onlisp:analyze-body body)
    (unless doc
    `(progn
       (defmethod ,name (,(first parameter))
       'doc
       ,@dec
       ;; Push an invocation record onto the call stack.
       (push (get-invocation ,(caar parameter) ',name) *CALL-STACK*)
       ;; Explicit trap for unbound-slot exception. A dependency will
       ;; have been set up (because our :before specialization of
       ;; slot-value-using-class will have executed) such that
       ;; population of the slot will result in the necessary
       ;; constraints re-running.
       (catch 'constraint-exit
 (handler-bind
     ((error #'Constraint-Condition-Handler))
   (Ensuring-Instances
    (:key (first *CALL-STACK*) :parent ,(caar parameter))
       ;; Call the dash function
    (,(intern (format nil "-~a" name)) ,(caar parameter)))))
       ;; Pop the call stack.
       (pop *CALL-STACK*)
 nil) ; does not return a useful value.
       ;; Define the dash function.  Since constraints return values are
       ;; thrown away by design, we define a delegator for traceability.
       (defmethod ,(intern (format nil "-~a" name)) (,(first parameter))
 ,@bod))))

(defmacro defquery (name parameters &body body)
  "This macro provides an interface to a COSI object which ensures that
constraints are propagated before the the object is referenced

The macro-results in a method being defined using the name parameters and body
supplied.

As a matter of style a defquery should not do any computation but should only
access a COSI object slot or COSI method.
  e.g. (defquery complete ((om rst-object-model))
          (complete (visit-object om)))

The encapsulation for set-propagating and get-propagating is used to
prevent an infinite loop in propagation."
```

```
  (multiple-value-bind (dec doc bod) (onlisp:analyze-body body)
    (unless doc
      (warn (format nil "No documentation found for query ~A." name)))
    `(defmethod ,name ,parameters
,doc
       ,@dec
       (cond ((and (not (get-propagating))
   *propagate-constraints*)
      (unwind-protect
  (progn
    (set-propagating t)
    (propagate))
(set-propagating nil))))
       (let ((return-value nil))
 (handler-case
     (setf return-value (multiple-value-list (progn ,@bod)))
   (unbound-slot (message)
     (declare (ignore message))))
 (apply #'values return-value)))))
```

References

[1] Stephen A. Bloch. Scheme and Java in the First Year. In *Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 157–165. The Consortium for Computing in Small Colleges, 2000.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[3] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[4] Erann Gat. Point of view: Lisp as an alternative to Java. *Intelligence*, 11(4):21–24, 2000.

[5] Paul Graham. *On Lisp*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[6] Paul Graham. Remarks of Carl de Marcken, January 2001. Email correspondence.

[7] Robert Harper. Programming languages: Theory and practice. working draft. Textbook in progress, 2003.

[8] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York, NY, 1991. ISBN 0-07-032813-7.

[9] Guy Steele Jr. *Common Lisp the Language.* Digital Press, second edition, 1990.

[10] Sonya Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[11] David A. Moon. Object-oriented programming with flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[12] Lutz Prechelt. Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, 1999.

[13] Yen-Ping Shan, Tom Cargill, Brad Cox, William Cook, Mary Loomis, and Alan Snyder. Is multiple inheritance essential to OOP? (panel). In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 360–363. ACM Press, 1993.

[14] Marvin V. Zelkowitz Terrence W. Pratt. *Programming Languages: Design and Implementation, 3rd ed.* Prentice-Hall, Inc., 1996.

LISP and Productivity in the Development of Complex Systems

Robert Hawkins

PSDB

Tuesday, November 19, 2002

Abstract

For the development of complex software systems, the combination of an interactive top-level and debugging environment and the speed of truly compiled code provided by LISP provides development capabilities that are, in my opinion, superior to any other development environment. In this position paper, I will provide examples of how these capabilities provide increased efficiency, allowing for quick turn-around of code for users that I argue would not be possible in any other environment. .

1 Introduction

For more than 40 years, Lisp has often been used for the development of complex systems. Some of the primary reasons behind this are the advanced features and expressiveness of the language, and its unique productivity features that enable the combination of interactive coding and debugging and the speed of compiled code in a seamless fashion.

In this position paper I will first explore in depth the latter set of features, by providing personal examples of why that particular set of features make Lisp unparalleled for the development of complex systems. Following the discussion of productivity features, I will address some technical issues with Lisp, and discuss IDE progress for Java.

2 Productivity Features

The lisp development environment supports a nearly unique combination of two features – being both interactive and compiled – which provides a level of developer productivity and run-time speed that is rarely seen in other environments. The interactive environment and its *persistent state* are where I will focus the majority of my discussion. Having this persistent state in Lisp means that not only can one interactively define code, but the code can be applied to the existing object model in context, including redefinitions. There are 3 primary roles I will discuss that this environment provides: Interactivity, “Fix and Restart” and Rapid Prototyping.

2.1 Interactivity

The interactive command interpreter provides coding, debugging and tracing features in Lisp that increase productivity and facilitate learning at any level. The interactive interpreter provides the ability to interact directly with any of the instances of objects within the model at any time due to the persistent state of the coding environment. Because of this feature, the developer can learn by example, prototype, test code, and debug in a superior fashion than that of other environments.

2.1.1 Examples

Tracing Though debuggers for non-interactive languages provide tracing capabilities, I feel that through the interpreter, Lisp provides a more full-featured ability to trace. One reason for my position is the interpreter provides the debugger, tracer, profilers all within the same seamless environment, and all in the same language. These tools are always available and can be turned on at any point, even after an application has begun processing.

This means that if the developer testing a large set of data and output appears suspicious, a trace can be turned on to help illuminate the problem without restarting. Another feature that makes tracing in the interactive environment so useful is that the actual trace output can be used with little or no modification to interactively call the method in question. The developer can then recreate the method call after breaking the code and run the problem code as is, or with parameters slightly changed.

Lisp’s trace facility also allows extensive parameterization of the trace itself so that its scope can be narrowed. For example, further information can be printed upon entry or exit to the function, and conditional breaks can be placed.

Dynamic code definition Frequently when developing a complex system, it is not only advantageous to be able to look at the object model, but to be able to define functions to test features of the data. Because debugging, running and coding can all be combined within the interactive environment of lisp, it is trivial to write code to test features of the data – count the number of objects with feature X, etc.

For example, with the SIRTF version of SPIKE, I have approximately 10,000 visits for a 1-year schedule. When designing features of the long range planner, I frequently rely on the ability to throw together a bit of code to test anything from the number of visits which use a particular instrument to more detailed features such as how many visits can schedule in a particular time window if a particular instrument gets assigned that window.

2.1.2 Discussion

The interactive features of lisp and the coding style that it supports are intrinsic to the style of coding which I use. In any non-interactive environment, I would be handicapped trying to work without these missing features. Conversely, in a non-compiled environment, I would be handicapped by the compute time restrictions when testing large sets of data. Thus, Lisp is uniquely suited to the combination of style of coding and application domain that underpins my requirements for a programming environment.

2.2 Fix and restart

From where I sit, there is little that rivals the ability to "fix and restart" when it comes to productivity, particularly when developing large and complex systems.

Large sets of data are the norm in scheduling algorithm development. The routine process of loading these sets of data can take on the order of 10 minutes to an hour; scheduling runs can take hours or even days. The ability to change code on the fly and restart if an initial coding was wrong, without the requirement to plod through the build-compile-run loop and then regenerate state can be **crucial** to rapid development and productivity.

2.2.1 Examples

Scheduling algorithm development for SIRTF The development of algorithms for long-range planning/scheduling is a domain and data dependant process. Prototyping these algorithms accurately require one to frequently test on **real** data sets and something close to operational volumes of data. This is necessary due to the fact that these are 1 year "long range" scheduling algorithms, thus a majority of prototyping and testing would not be accurate on smaller datasets. Frequently, when coding new scheduling rules for SIRTF, the new rule may not actually be applied until a final step in the scheduling algorithm. It may take 2 hours to get to this step. If there is a bug in the original code, I should be able to just fix it and restart (and perhaps iterate those two steps) with lisp and the finish the run. If I was using a non-interactive language, I would (in many situations) have to enter a debugger to hope to find the problem, attempt to fix the source, recompile, reload the application and the data, restart the scheduler, and wait for it to reach the new rule to see if the fix was successful. This means that iterations to repair bugs could be on the order of hours rather than minutes.

Repairing Runtime Crashes Should a bug occur in released code, it is possible that a crash would happen during testing or a scheduling run. With the interactive "fix and restart" ability of lisp, if a developer is available (or the lisp image is dumped) the problem can be investigated and solved in situ, potentially allowing the run to complete. Conversely, in a non-interactive system, the developer might be required to re-run to elicit debug information, then attempt the fix, followed by another run for testing and so on. An example of how this interactive feature saves time can be found in Zimmerman and Asson (2002):

"This included processing many visits and was expected to run for six hours. Spike crashed at the very end of the run, just minutes from completion. The debugger told us where the crash occurred and we understood exactly what had gone wrong. Instead of having to fix the bug and rerun the entire process, we were able to load the software fix right into the LISP interface and restart from that point. The process finished in a few minutes and we saved six hours."

2.2.2 Discussion

The capability of lisp to change code definitions within the persistent state of the complex system can provide an invaluable savings in turnaround time for the developer or the user. It is very difficult to directly measure this benefit, but through some of these examples, it should be apparent that the savings in developer and user time could be immense. This savings grows at least proportionately with the complexity of the system being modeled.

2.3 Rapid Prototyping and Incremental Development

Rapid prototyping/incremental development is another capability supported by the dynamic interpreted environment of lisp that enhances productivity. One feature that supports this is the capability to dynamically (re)define methods and modify classes either at the command line or in a buffer and compile them directly into the existing image, applying them to extant data. This ability can be useful both in the development of complex algorithms and for things as seemingly trivial as reporting.

2.3.1 Examples

Scheduling Again, in the context of building a scheduling algorithm to use upon a large body of data, the ability to rapidly prototype is a key factor in how Lisp makes me more productive. In the situation describe above in the discussion on fix and restart, if a new algorithm for a sub-step is called for, once a problem occurs, it is possible to interactively write and test that function inline within the debugging context (as the debugger is part of and contains all of the power of the language). This allows that iterative process to be reduced even further.

Reporting Rapid prototyping can be especially useful when formatting output such as textual reports and plots. Activities like getting the alignment of a column just right, placing a hash mark in the right and ordering output can most easily be accomplished by iterating – try a value, see what happens, try again. The layout of output products may seem like a mundane task, but it is often critical for users. With an interactive environment, this is trivial, and can be accomplished in a short time. This would not be the case in a strictly compiled environment.

A user of FUSE Spike asked for a new plot for use in a presentation. I received the request at around 3:30 p.m. on a day I planned to leave at 4:30 p.m. With a short flurry of emails, we were able to specify the parameters of the new plot in less than half an hour, and I was able to prototype an output in another 20 minutes, and got out a set of patch files to the users in about 20 more minutes. I left a bit later than I had planned that day, but I believe that the users never would have gotten that plot that day without the rapid prototyping ability of Lisp.

A similar example of rapid prototyping a specialized output product for HST SPIKE is given in Zimmerman and Asson (2002), pg. 3, where a user requested a specialized report for a large number of visits, which was not currently supported, and not likely to be needed in the future:

"The (solution) approach was to write three LISP functions and give them to him in a file. He loaded that file into his currently running Spike process and ran the one interface function that produced his report. He did not need to restart the system or reload his proposal. Note that loading a complicated, 300+ visit proposal may take a long time. We were able to deliver a reporting capability in approximately 20 minutes without the need for a restart or reload. We also did not introduce code into the system that might not be useful for anyone else. Finally, we did not need to create a new image, which is time consuming and wastes disk space."

2.3.2 Discussion

Clearly, rapid prototyping can be of use in many situations. These examples serve to highlight only a subset of the instances in which I have seen the utility of this feature. The persistent state and dynamic update capabilities of lisp again show their strength here, as evidenced by the final example, in which a developer rapidly prototyped something at his workstation, emailed the set of patches to the user, who was able to compile the code and use it immediately on the existing data he had in his image without the need for a costly (in terms of time) reload.

3 Technical Discussion

3.1 GUI Support

The major technical failing of LISP is the lack of an up-to-date multi-platform GUI. Although LISP has CLIM for GUI support, its features are fairly behind-the times, non-standard and extremely poorly documented in comparison to the rest of LISP's features. As such, I do not recommend it as a GUI development platform for the future. Thus, if we are to use LISP as our development language of choice, we are required to implement a multi-language solution. Though this might seem to have some disadvantages (not the least of which would be the inability to use LISP's productivity features in GUI development), there are ameliorating factors.

Our LISP vendor, Franz, has recently incorporated the "jlinker" package which provides a reasonably simple interface to Java which enables anything from a simplified protocol for implementing a Java/Lisp link to the ability to write Java code directly inline with lisp code. Our LISP vendor provides excellent socket and CORBA packages as well.

Additionally, there are benefits to enhanced modularity between GUI and scheduling code. GUI technology changes far more rapidly then is necessary for base application code. SPIKE, for example, has already gone through at least 3 GUI projects (and is beginning a fourth) during its existence. In addition, GUIs tend to need more specialization for individual missions than application source. This modularity could be enforced in one language, of course, but a multi-language solution, when done right, should theoretically allow "plug and play" GUI implementations in the future, lending a fairly large advantage to having developed the GUI in another language.

3.2 Language Level

The "Language Level" of a language also affects productivity. Though this productivity is not everything in a project (coding may only amount to 30-50% of a project), it **is** important. Language Level in general translates to the number of program statements required to code one Function Point. The language Level of assembler is 1.0, C is 2.5, for example. The Common Lisp Object System (CLOS) is rated at 15 with C++ and Java at 6, with no data for Python, though it is probably close to CLOS. Some empirical studies show CLOS/Lisp solutions to similar problems taking 20 to 50 percent of the number of statements as Java and C++.

An example of the succinctness and speed of Lisp can be seen by following Peter Norvig's article "Lisp as an Alternative to Java" (Norvig, 2001). In this article, Norvig coded a solution to a problem that was given in a study of 38 C, C++ and Java developers who were asked to implement the problem. (Prechelt, 1999) In addition, Gat (Gat, 2000) had done a follow-up study where he asked programmers to write the same test program in Lisp. Finally, Prechelt also did a follow-up with "scripting" languages, including Python. (Prechelt, 2000) The final results were that the Java and Python solutions were slower than those in C, but the Lisp solutions were on par with the C versions (the Lisp average was faster, but the fastest C solution was the fastest of all solutions.) The results in the realm of productivity were (Norvig himself solved the problem in 2 hours, with 42 lines of code):

Language	Coding Time	Code Length
C/C++	3-25	150-510
Java	4-63	100-614
Lisp	2-8.5	51-182
Python	1.5-6	60-225

A description of an individual C++ developer attempting to match Norvig and the expressiveness of Lisp can be seen in Corfman (2002). This developer had run across Norvig's article above and attempted the challenge himself. His initial solution was 220 lines long and it took him 8 hours to code. After iterating a few times and modifying the C Standard Template Library, to support the problem, he was able to get his code down to 79 lines (48 not including #includes and braces). Although the developer did not "convert" to lisp, it is obvious that he sees many of the benefits and beauties of lisp (and is never able to match the succinctness that the expressiveness of lisp provides, despite a number of iterations.)

3.3 Other Languages and Development Environments

I have focused mostly on the productivity aspects of LISP, implying that the other languages are deficient in these areas. In defense of the rapid development and improvement of IDEs for Java and C, we should explore the progress that **is** being made, particularly by Java IDEs in this realm. I have not been able to allocate as much time to this project I needed to allow me to investigate this issue as fully as I originally desired, but I want to at least touch on some particular advances that I have seen that look promising.

3.3.1 Codeguide

Codeguide is a Java IDE that is used extensively by Java developers at STSCI. Codeguide is a quality product that is being rapidly updated (two major releases in less than a year), and which is incorporating many useful productivity features, some of which are similar to lisp.

On-the-fly compilation

Codeguide version 5 has a newly incorporated feature dubbebd "on-thy-fly compilation". While I have not yet had a chance to test this feature, its description sounds promising. Codeguide (2002):

"CodeGuide offers you incremental on-the-fly ("instant") compilation. This feature is unique in the world. CodeGuide compiles your programs as you type, showing you errors in your program instantly. There are no compilation times when you want to start your program because the program has already been compiled in the background. This changes the ordinary "edit-compile-edit-compile-...-run" cycle to just "edit-run"."

This feature is primarily of use in that it reduces the edit-compile-run-load cycle to edit-run-load, at least. This reduces that part of the overhead of developing in compiled languages.

HotSwap dynamic class replacement

"Replace code while the debugger is running without the need to restart the application. The debugger will execute the replaced code. Together with the instant compilation this feature offers you unbeatable turn-around times." Codeguide (2002):

This feature also shows potential. The documentation is sparse on the feature, and again, I have not had a chance to test it. My primary concern is to how much of the application and its data must be renewed due to the replacement. For example, if I had an instance of a visit and I redefined the visit class, would the instance still exist, or would I have to reload. As I have mentioned, in complex systems like SPIKE, it is frequently the case that reloading the model itself is more time consuming then recompiling and restarting the application itself. If this feature invalidates the data model, then its utility is severely lessened.

3.3.2 VisualAge

VisualAge for Java by IBM is another IDE that I have investigated, due to the fact that it is modeled after a Smalltalk environment. Prior to coming to the institute, I was a Smalltalk developer, so I felt that this IDE might appeal to me.

Scrapbook

VisualAge for Java has the concept of a "Scrapbook" in which one can execute code fragments and test them inline. This provides at least some of the "interactivity" features of Common Lisp that makes it so easy to learn and to prototype in. Unfortunately, this scrapbook is not persistent. If a complex sequence is run, and a further step is desired, the step must be added, and then the original code must be executed again. Code also cannot be tested on a loaded model.

Debugging

Because VisualAge is a product developed by a Smalltalk provider, and as such has a lot of the look and feel of a Smalltalk development environment, the debugging features of VisualAge look incredibly promising to me, as a former Smalltalk developer. I feel that Smalltalk has an equally strong set of productivity features as lisp (plus a slightly more powerful debugger with a source enabled debug stack), but due to the fact it is not compiled and thus slower, SmallTalk was not considered in this language study. A debugging environment **like** Smalltalk but for Java, is very enticing, however.

Run-time code changes

VisualAge also incorporates a capability to change code on the fly while an applet is running. Again, I have not yet thoroughly tested this feature. It is described in IBM's literature as follows, Visualage (2002):

"When you are running an applet in the IDE's Applet Viewer, you can edit the applet's source code and then see the results of your changes immediately."

My reservations discussed above with regard to Codeguide's similar feature also apply here. The implication of the statement "see the results of your changes immediately" may be that my reservations are unfounded, however.

3.3.3 What's Missing

As far as I have seen, none of these other development environments are providing the core interactive feature of data persistence that makes the lisp environment so powerful. Although I find myself enthused by the advancements other IDEs are making towards providing the productivity features that lisp has long had, I am skeptical about whether any of these environments will make this final step.

Overall, I am quite impressed with the IDEs I have seen for Java, and will be investigating them in a more thorough manner, as we will be developing Java GUIs for SPIKE.

3.4 Conclusion

In conclusion, for my style of development, it all boils down to the persistence of the data model of the complex system. The features of Lisp (and perhaps other interactive languages) that support this provide unparalleled capabilities to rapidly prototype, debug and support users. The combination of developer productivity and compiled run-time speed in Lisp is nearly unique. The efficiency that is fostered by this feature set in the individual developer and the organization as a whole is hard to quantify but must be considered when evaluating languages for the development of complex systems.

3.5 References

Corfman, B.,"C++ vs. Lisp," August, 2002.

IBM, "Visualage for Java online help", 2002.

Norvig, P., "Lisp as an Alternative to Java" (Webpage), 2001.

Omnicore, "Features Included in CodeGuide 5.0",

Pittman, K. "Accelerating Hindsight: Lisp as a Vehicle for Rapid Prototyping," In *Lisp Pointers*, Volume VII, Number 1-2, January-June 1994.

Prechelt, L., "Comparing Java vs. C/C++ Efficiency Issues to Interpersonal Issues" In Communications of the ACM, October, 1999.

Prechelt, L., "An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-
Technical Report, March, 2000.

Zimmerman-Foor, L. and Asson, D.J., "Spike: A Dynamic Interactive Component In a Human-Computer I
Presented at The Third International Workshop on Planning and Scheduling for Space, October, 2002.

The Position of Chris Sontag

My language recommendation for future P&S code is Java or C++, depending on the problem domain.

This working group has been interesting to me in that it has effectively brought into a focus a full set of programming language features, many with which I consider myself very familiar, but some with which I am not. For the most part, my own learning has had to do with features of Lisp that are simply not present in other languages. The expressiveness and flexibility of Lisp obviously make it a very powerful tool, and I would like someday to spend some time exploring it more thoroughly. In comparison to some of the other languages in our study however, Lisp suffers greatly. In part this is due to metrics such as support for: encapsulation (member access control, private data, etc), 'const'-ness, graphical user interfaces, and others, but I find the largest detraction to be the narrow usage of the language. I cannot, in good conscience, suggest that new projects be written in a language that requires newly hired developers to be taught first before they can begin to be productive. Training is very often a good investment, but the existence of a requirement to train represents a potential danger to a project over its lifetime. The real concern is the size of the pool of acceptable candidates at hiring time. Knowledge of the language (or willingness to learn it) is but one factor that goes into that pool size. It is impossible to measure the impact to a project that comes from using a newly-trained fresh college graduate, instead of a seasoned developer who is fluent in the language. It is impossible to measure but it is easy to imagine. With a language like Lisp, there are very simply a great deal less seasoned developers available. It is the unfortunate responsibility of this working group to concern ourselves with such 'non-technical' issues. And, since one can probably predict that this trend will continue, the concern over the number of available Lisp developers for hire becomes that much more important for future projects. In addition, I do not believe that simply training new hires in Lisp is even a reasonable 'backup plan', as most projects will require at least a few developers that can start with an advanced knowledge in their craft.

I find C++ and Java both to be sufficiently flexible/expressive to be able to handle complex projects. I refer the reader to the language feature table that this working group has created. Given that fact, I propose that either of these two languages are preferable over Lisp for future Planning and Scheduling problem domains. As for selection between C++ and Java, I state the overly simple summary that, in the absence of extenuating circumstances (legacy C/C++ code), and when Java's performance is acceptable for the problem at hand, the best choice at this time (mid 2002) is Java.

The working group also considered C# and Python. C# has great attraction due to its current similarity to Java in features, its expected performance improvements on the Windows platform, and its wide interoperability. However, C#'s current level of maturity (infancy) and the fact that it is, by its very design, tied to a single platform, are obstacles that destroy its potential here. Python is also an interesting candidate. While no single issue jumps out as a major detraction to Python (except possibly performance), Python in general ranked lower than the other candidates in enough places to warrant its removal from serious consideration - again I refer the reader to the language feature table. It should be noted however, that Python admittedly has excellent potential as a 'glue' language for relating software components in this domain.

The Position of Scott Speck

The Planning and Scheduling Language Study Group was chartered with the task of comparing the suitability of several high-level software languages for use in developing Planning and Scheduling software for the Next Generation Space Telescope (NGST). The list of languages under consideration include C++, Java, Lisp, and Python.

It is my position, after numerous group discussions about the relevant technical and non-technical issues involved in evaluating the features of the candidate languages, that Java is the best choice, given my knowledge of the design requirements for planning and scheduling for the NGST ground system.

This choice is contingent upon my understanding of the requirements of the NGST P&S system, and it is handicapped, admittedly, by my lack of experience with Lisp and its potentially rich set of capabilities.

My reasons for choosing Java over C++, Lisp, and Python are as follows:

1) Java, being, in essence, an interpreted language, is slower for numerically intensive computations than C++ or Lisp. However, it should be kept in mind that NGST P&S for NGST will not require numerically intensive calculations to nearly the degree currently performed by SPSS software with HST. Most of the intensive orbit-dependent calculations for NGST will be performed by "flight software", and will be beyond the domain of planning and scheduling. As a result, numerical computation speed for NGST P&S is of low to moderate impact, and Java would thus be "fast enough" for the task at hand.

2) Java adequately supports object oriented programming. It is clear that the current paradigm for software engineering is object oriented in nature. All of the languages under consideration support this paradigm, including Java.

3) Java has better "safety features" than C++ or Lisp. This includes garbage collection, to help minimize the effects of memory leaks. With Java, one is much less likely to encounter bugs which arise due to improper pointer manipulation and arithmetic than with C++. In addition, Java supports data encapsulation within objects, strengthening its support of the OO paradigm as well as enhancing the integrity of object data contents.

4) Java has a very rich set of libraries which, moreso than the other languages, prevent one from having to reinvent the wheel. GUI support, graphics, networking, multi-threaded programming, shared resources, and many other features are straightforward for use in Java, largely through the use of pre-existing libraries which already implement these capabilities.

5) Java produces bytecodes which are portable across platform architectures. This portability could allow P&S to be run easily on a variety of platforms, without the need to provide intensive, separate support for each platform. The other languages under consideration do not provide this degree of platform independence.

6) Java is a very popular and attractive language to software developers, over a wide domain of programming applications. This will aid in attracting and keeping a good programming staff for P&S, as well as maintainers of the system, as staffing evolves over NGST's lifetime. I feel that Java currently enjoys more popularity and is deemed more attractive than the other languages, by programmers in general.

In summary, I feel that Java provides good OO support, a rich set of programming libraries, good GUI support, relative platform independence, and sufficient numerical efficiency, as well as providing lower hiring/training costs (because so many software developers have experience with and prefer to program in Java) than the other language options. With all of its virtues taken together, Java provides the best ensemble of features and popularity, given the problem domain.

The Position of Frank Tanner

1 Executive Summary

The language subcommittee of the Future of Planning and Scheduling Working Group was chartered with determining the most appropriate language for implementing a planning and scheduling system at STScI. My position is that the Java programming language would be the most appropriate choice for implementing an entirely new P&S system.

The arguments that support this decision are outlined in detail in this paper. The high-level points are:

The non-technical features of Java outweigh any technical limitations of Java

The Common Lisp (CL) programming language alone cannot fully implement a P&S system since it lacks a robust, multi-platform GUI. Therefore, a multiple language solution would be required.

Java is a more forward looking, innovative language because of the strong Java community supporting it

CL does not fit into the ESS culture

2 Notes & Disclaimers

First, I will not try to state that Java is a superior programming language than the other candidates. From a strict computer science point of view, CL is the superior programming language evaluated in this study. However, as this paper will attempt to prove, factors other than the programming language features make Java a better choice for implementing a P&S system at STScI.

The focus of this paper will be on Java and CL. I do not have the expertise to fully discuss Python. Also, if in the design of a new P&S system, raw speed and performance is a critical factor, than this entire discussion is moot and C++ should be strongly considered. In the absence of a strong performance requirement, C++ can be dismissed in favor of Java because of Java's extra language functionality and feature capabilities.

For the most part, hard statistics are unavailable for the majority of the discussion points the subcommittee evaluated. Subcommittee members relied on their personal experience and judgment as software developers to make their decisions.

3 Discussion

The bottom line in any programming language evaluation should be which programming language decreases development costs and increases employee productivity over the lifetime of a project. Therefore, this paper will focus on this as the primary choice criterion.

3.1 Non-Technical Features

At the heart of this discussion will be an analysis of the non-technical features of Java and Lisp that make Java a more appropriate choice for writing a P&S system at STScI.

3.2 Multiple Languages

CL lacks robust, multiple platform GUI support. Allegro CL does have a good GUI for Windows. However, since the primary audience for our P&S system is still on Solaris, Windows GUI support is not important.

Given that CL lacks this support, a multiple language implementation would be required. In my experience, developing multiple-language applications increases costs significantly. Although difficult to quantify, the following aspects of multiple language implementations increase costs:

Increased training costs

Increased deployment costs

Increased performance impacts (i.e. IPC has communication overheads that single language implementations do not have)

Interface Control Documents (ICDs) are time consuming to keep current and incur extra maintenance costs

Increased debugging costs (both developing a debugging infrastructure and determining where bugs reside)

3.3 ESS Culture

3.3.1 Lisp Justification

This point is more of a resignation to ESS's culture rather than a point about a programming language. Lisp is a misunderstood language in ESS software development. Its extensive capabilities and robust features are very desirable in a programming language. With that said, ESS does not have the infrastructure to support Lisp. Over the past couple of years, every time a new software development project is brought up, PDT/PSDB must justify its use of Lisp. Whereas, if Java or C++ was chosen, I would propose that a small 2-3 paragraph write-up would be required to justify our choice. Any time that Lisp is even proposed, extensive research and discussions must take place in order to support that decision.

3.3.2 Collaboration

Further, Lisp isolates the PSDB team from the rest of the institute. This is an extension to the popularity argument outlined above. When a single branch of an organization adopts a technology that is not utilized in the rest of the organization, the branch becomes an isolated unit. This, at times, makes it difficult for other branches to interact and work with PSDB. Having a more ubiquitous technology such as Java would better foster collaboration and teamwork across ESS Branches.

Beating The Averages

Paul Graham

www.paulgraham.com/avg.html
Included with permission

(This article is based on a talk given at the Franz Developer Symposium in Cambridge, MA, on March 25, 2001.)

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

A lot of people could have been having this idea at the same time, of course, but as far as I know, Viaweb was the first Web-based application program. It seemed such a novel idea to us that we named the company after it: Viaweb, because our software worked via the Web, instead of running on your desktop computer.

Another unusual thing about this software was that it was written primarily in a programming language called Lisp.[1] It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. Lisp gave us a great advantage over competitors using less powerful languages.

1 The Secret Weapon

Eric Raymond has written an essay called "How to Become a Hacker," and in it, among other things, he tells would-be hackers what languages they should learn. He suggests starting with Python and Java, because they are easy to learn. The serious hacker will also want to learn C, in order to hack Unix, and

[1]Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The first version was mostly Lisp, because the ordering system was small. Later we added two more modules, an image generator written in C, and a back-office manager written mostly in Perl.

Perl for system administration and cgi scripts. Finally, the truly serious hacker should consider learning Lisp:

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will improve your mind, and make you a better writer in languages you do want to use, like English.

But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

So if Lisp makes you a better programmer, like he says, why wouldn't you want to use it? If a painter were offered a brush that would make him a better painter, it seems to me that he would want to use it in all his paintings, wouldn't he? I'm not trying to make fun of Eric Raymond here. On the whole, his advice is good. What he says about Lisp is pretty much the conventional wisdom. But there is a contradiction in the conventional wisdom: Lisp will make you a better programmer, and yet you won't use it.

Why not? Programming languages are just tools, after all. If Lisp really does yield better programs, you should use it. And if it doesn't, then who needs it?

This is not just a theoretical question. Software is a very competitive business, prone to natural monopolies. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. In a startup, if you bet on the wrong technology, your competitors will crush you.

Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best.

This is especially true in a startup. In a big company, you can do what all the other big companies are doing. But a startup can't do what all the other startups do. I don't think a lot of people realize this, even in startups.

The average big company grows at about ten percent a year. So if you're running a big company and you do everything the way the average big company does it, you can expect to do as well as the average big company– that is, to grow about ten percent a year.

The same thing will happen if you're running a startup, of course. If you do everything the way the average startup does it, you should expect average performance. The problem here is, average performance means that you'll go out of business. The survival rate for startups is way less than fifty percent. So if you're running a startup, you had better be doing something odd. If not, you're in trouble.

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run our your own servers, you can use any language you want. When you're writing desktop software, there's a strong bias toward writing applications in the same language as the operating system. Ten years ago, writing applications meant writing applications in C. But with Web-based software, especially when you have the source code of both the language and the operating system, you can use whatever language you want.

This new freedom is a double-edged sword, however. Now that you can use any language, you have to think about which one to use. Companies that try to pretend nothing has changed risk finding that their competitors do not.

If you can use any language, which do you use? We chose Lisp. For one thing, it was obvious that rapid development would be important in this market. We were all starting from scratch, so a company that could get new features done before its competitors would have a big advantage. We knew Lisp was a really good language for writing software quickly, and server-based applications magnify the effect of rapid development, because you can release software the minute it's done.

If other companies didn't want to use Lisp, so much the better. It might give us a technological edge, and we needed all the help we could get. When we started Viaweb, we had no experience in business. We didn't know anything about marketing, or hiring people, or raising money, or getting customers. Neither of us had ever even had what you would call a real job. The only thing we were good at was writing software. We hoped that would save us. Any advantage we could get in the software department, we would take.

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.

What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

It must have seemed to our competitors that we had some kind of secret weapon– that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.

When I was about nine I happened to get hold of a copy of The Day of the Jackal, by Frederick Forsyth. The main character is an assassin who is hired to kill the president of France. The assassin has to get past the police to get up to an apartment that overlooks the president's route. He walks right by them, dressed up as an old man on crutches, and they never suspect him.

Our secret weapon was similar. We wrote our software in a weird AI language, with a bizarre syntax full of parentheses. For years it had annoyed me to hear Lisp described that way. But now it worked to our advantage. In business, there is nothing more valuable than a technical advantage your competitors don't understand. In business, as in war, surprise is worth as much as force.

And so, I'm a little embarrassed to say, I never said anything publicly about Lisp while we were working on Viaweb. We never mentioned it to the press, and if you searched for Lisp on our Web site, all you'd find were the titles of two books in my bio. This was no accident. A startup should give its competitors as little information as possible. If they didn't know what language our software was written in, or didn't care, I wanted to keep it that way.[2]

The people who understood our technology best were the customers. They didn't care what language Viaweb was written in either, but they noticed that it worked really well. It let them build great looking online stores literally in

[2]Robert Morris says that I didn't need to be secretive, because even if our competitors had known we were using Lisp, they wouldn't have understood why: "If they were that smart they'd already be programming in Lisp."

minutes. And so, by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. It's one of the more profitable pieces of Yahoo, and the stores built with it are the foundation of Yahoo Shopping. I left Yahoo in 1999, so I don't know exactly how many users they have now, but the last I heard there were about 14,000.

People sometimes ask me if Yahoo Store still uses Lisp. Yes, all the Lisp code is still there. Yahoo has server-side software written in all five of the languages Eric Raymond recommends to hackers.

2 The Blub Paradox

What's so great about Lisp? And if Lisp is so great, why doesn't everyone use it? These sound like rhetorical questions, but actually they have straightforward answers. Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower. Of course, both these answers need explaining.

I'll begin with a shockingly controversial statement: programming languages vary in power.

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers.

Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one. [3]

[3] All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't

There are many exceptions to this rule. If you're writing a program that has to work very closely with a program written in a certain language, it might be a good idea to write the new program in the same language. If you're writing a program that only has to do something very simple, like number crunching or bit manipulation, you may as well use a less abstract language, especially since it may be slightly faster. And if you're writing a short, throwaway program, you may be better off just using whatever language has the best library functions for the task. But in general, for application software, you want to be using the most powerful (reasonably efficient) language you can get, and using anything else is a mistake, of exactly the same kind, though possibly in a lesser degree, as programming in machine language.

You can see that machine language is very low level. But, at least as a kind of social convention, high-level languages are often all treated as equivalent. They're not. Technically the term "high-level language" doesn't mean anything very definite. There's no dividing line with machine languages on one side and all the high-level languages on the other. Languages fall along a continuum [4] of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power.

Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.

Or how about Perl 4? Between Perl 4 and Perl 5, lexical closures got added to the language. Most Perl hackers would agree that Perl 5 is more powerful than Perl 4. But once you've admitted that, you've admitted that one high level language can be more powerful than another. And it follows inexorably that, except in special cases, you ought to use the most powerful you can get.

This idea is rarely followed to its conclusion, though. After a certain age, programmers rarely switch languages voluntarily. Whatever language people happen to be used to, they tend to consider just good enough.

Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abtractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.

make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.

[4]Note to nerds: or possibly a lattice, narrowing toward the top; it's not the shape that matters here but the idea that there is at least a partial order.

And in fact, our hypothetical Blub programmer wouldn't use either of them. Of course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).

As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y.

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can't trust the opinions of the others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

I know this from my own experience, as a high school kid writing programs in Basic. That language didn't even support recursion. It's hard to imagine writing programs without using recursion, but I didn't miss it at the time. I thought in Basic. And I was a whiz at it. Master of all I surveyed.

The five languages that Eric Raymond recommends to hackers fall at various points on the power continuum. Where they fall relative to one another is a sensitive topic. What I will say is that I think Lisp is at the top. And to support this claim I'll tell you about one of the things I find missing when I look at the other four languages. How can you get anything done in them, I think, without z? And one of the biggest zs, for me, is macros. [5]

Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different. To the Blub programmer, Lisp code looks weird. But those parentheses are there for a reason. They are the outward evidence of a fundamental difference

[5]It is a bit misleading to treat macros as a separate feature. In practice their usefulness is greatly enhanced by other Lisp features like lexical closures and rest parameters.

between Lisp and other languages.

Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp. It would be convenient here if I could give an example of a powerful macro, and say there! how about that? But if I did, it would just look like gibberish to someone who didn't know Lisp; there isn't room here to explain everything you'd need to know to understand what it meant. In Ansi Common Lisp I tried to move things along as fast as I could, and even so I didn't get to macros until page 160.

But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25program is doing things that you can't easily do in any other language. However skeptical the Blub programmer might be about my claims for the mysterious powers of Lisp, this ought to make him curious. We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection. I encourage you to follow that thread. There may be more to that old man hobbling along on his crutches than meets the eye.

3 Aikido for Startups

But I don't expect to convince anyone (over 25) to go out and learn Lisp. The purpose of this article is not to change anyone's mind, but to reassure people already interested in using Lisp– people who know that Lisp is a powerful language, but worry because it isn't widely used. In a competitive situation, that's an advantage. Lisp's power is multiplied by the fact that your competitors don't get it.

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. And it's likely to. It's the nature of programming languages to make most people satisfied with whatever they currently use. Computer hardware changes so much faster than personal habits that programming practice is usually ten to twenty years behind the processor. At places like MIT they were writing programs in high-level languages in the early 1960s, but many companies continued to write code in machine language well into the 1980s. I bet a lot of people continued to write machine language until the processor, like a bartender eager to close up and go home, finally kicked them out by switching to a risc instruction set.

Ordinarily technology changes fast. But programming languages are different: programming languages are not just technology, but what programmers think in. They're half technology and half religion. [6] And so the median language, meaning whatever language the median programmer uses, moves as slow as an iceberg. Garbage collection, introduced by Lisp in about 1960, is now widely considered to be a good thing. Runtime typing, ditto, is growing in popularity. Lexical closures, introduced by Lisp in the early 1970s, are now, just barely, on the radar screen. Macros, introduced by Lisp in the mid 1960s, are still terra incognita.

Obviously, the median language has enormous momentum. I'm not proposing that you can fight this powerful force. What I'm proposing is exactly the opposite: that, like a practitioner of Aikido, you can use it against your opponents.

If you work for a big company, this may not be easy. You will have a hard time convincing the pointy-haired boss to let you build things in Lisp, when he has just read in the paper that some other language is poised, like Ada was twenty years ago, to take over the world. But if you work for a startup that doesn't have pointy-haired bosses yet, you can, like we did, turn the Blub

[6] As a result, comparisons of programming languages either take the form of religious wars or undergraduate textbooks so determinedly neutral that they're really works of anthropology. People who value their peace, or want tenure, avoid the topic. But the question is only half a religious one; there is something there worth studying, especially if you want to design new languages.

paradox to your advantage: you can use technology that your competitors, glued immovably to the median language, will never be able to match.

If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings. Everything else on their site may be stock photos or the prose equivalent, but the job listings have to be specific about what they want, or they'll get the wrong candidates.

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening– that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

Revenge of The Nerds

Paul Graham

www.paulgraham.com/avg.html
Included with permission

May 2002

(This is an expanded version of the keynote lecture at the International ICAD User's Group conference in May 2002. It explains how a language developed in 1958 manages to be the most powerful available even today, what power is and when you need it, and why pointy-haired bosses (ideally, your competitors' pointy-haired bosses) deliberately ignore this issue.)

Note: In this talk by "Lisp", I mean the Lisp family of languages, including Common Lisp, Scheme, Emacs Lisp, EuLisp, Goo, Arc, etc.

Let me start by admitting that I don't know much about ICAD. I do know that it's written in Lisp, and in fact includes Lisp, in the sense that it lets users create and run Lisp programs.

It's fairly common for programs written in Lisp to include Lisp. Emacs does, and so does Yahoo Store. But if you think about it, that's kind of strange. How many programs written in C include C, in the sense that the user actually runs the C compiler while he's using the application? I can't think of any, unless you count Unix as an application. We're only a minute into this talk and already Lisp is looking kind of unusual.

Now, it is probably not news to any of you that Lisp is looking unusual. In fact, that was probably the first thing you noticed about it.

Believe it or not, there is a reason Lisp code looks so strange. Lisp doesn't look this way because it was designed by a bunch of pointy-headed academics. It was designed by pointy-headed academics, but they had hard-headed engineering reasons for making the syntax look so strange.

1 Are All Languages Equivalent?

In the software business there is an ongoing struggle between the pointy-headed academics, and another equally formidable force, the pointy-haired bosses. Everyone knows who the pointy-haired boss is, right? I think most people in the technology world not only recognize this cartoon character, but know the actual person in their company that he is modelled upon.

The pointy-haired boss miraculously combines two qualities that are common by themselves, but rarely seen together: (a) he knows nothing whatsoever about technology, and (b) he has very strong opinions about it.

Suppose, for example, you need to write a piece of software. The pointy-haired boss has no idea how this software has to work, and can't tell one programming language from another, and yet he knows what language you should write it in. Right: he thinks you should write it in Java.

Why does he think this? Let's take a look inside the brain of the pointy-haired boss. What he's thinking is something like this. Java is a standard. I know it must be, because I read about it in the press all the time. Since it is a standard, I won't get in trouble for using it. And that also means there will always be lots of Java programmers, so if the programmers working for me now quit, as programmers working for me mysteriously always do, I can easily replace them.

Well, this doesn't sound that unreasonable. But it's all based on one unspoken assumption, and that assumption turns out to be false. The pointy-haired boss believes that all programming languages are pretty much equivalent. If that were true, he would be right on target. If languages are all equivalent, sure, use whatever language everyone else is using.

But all languages are not equivalent, and I think I can prove this to you without even getting into the differences between them. If you asked the pointy-haired boss in 1992 what language software should be written in, he would have answered with as little hesitation as he does today. Software should be written in C++. But if languages are all equivalent, why should the pointy-haired boss's opinion ever change? In fact, why should the developers of Java have even bothered to create a new language?

Presumably, if you create a new language, it's because you think it's better in some way than what people already had. And in fact, Gosling makes it clear in the first Java white paper that Java was designed to fix some problems with C++. So there you have it: languages are not all equivalent. If you follow the trail through the pointy-haired boss's brain to Java and then back through Java's history to its origins, you end up holding an idea that contradicts the assumption you started with.

So, who's right? James Gosling, or the pointy-haired boss? Not surprisingly, Gosling is right. Some languages are better, for certain problems, than others. And you know, that raises some interesting questions. Java was designed to be better, for certain problems, than C++. What problems? When is Java better and when is C++? Are there situations where other languages are better than either of them?

Once you start considering this question, you have opened a real can of worms. If the pointy-haired boss had to think about the problem in its full complexity, it would make his brain explode. As long as he considers all languages equivalent, all he has to do is choose the one that seems to have the most momentum, and since that is more a question of fashion than technology, even he can probably get the right answer. But if languages vary, he suddenly has to solve two simultaneous equations, trying to find an optimal balance between two things he knows nothing about: the relative suitability of the twenty or so leading languages for the problem he needs to solve, and the odds of finding programmers, libraries, etc. for each. If that's what's on the other side of the door, it is no surprise that the pointy-haired boss doesn't want to open it.

The disadvantage of believing that all programming languages are equivalent is that it's not true. But the advantage is that it makes your life a lot simpler. And I think that's the main reason the idea is so widespread. It is a comfortable idea.

We know that Java must be pretty good, because it is the cool, new programming language. Or is it? If you look at the world of programming languages from a distance, it looks like Java is the latest thing. (From far enough away, all you can see is the large, flashing billboard paid for by Sun.) But if you look at this world up close, you find that there are degrees of coolness. Within the hacker subculture, there is another language called Perl that is considered a lot cooler than Java. Slashdot, for example, is generated by Perl. I don't think you would find those guys using Java Server Pages. But there is another, newer language, called Python, whose users tend to look down on Perl, and more waiting in the wings.

If you look at these languages in order, Java, Perl, Python, you notice an interesting pattern. At least, you notice this pattern if you are a Lisp hacker. Each one is progressively more like Lisp. Python copies even features that many Lisp hackers consider to be mistakes. You could translate simple Lisp programs into Python line for line. It's 2002, and programming languages have almost caught up with 1958.

2 Catching Up with Math

What I mean is that Lisp was first discovered by John McCarthy in 1958, and popular programming languages are only now catching up with the ideas he developed then.

Now, how could that be true? Isn't computer technology something that changes very rapidly? I mean, in 1958, computers were refrigerator-sized behemoths with the processing power of a wristwatch. How could any technology that old even be relevant, let alone superior to the latest developments?

I'll tell you how. It's because Lisp was not really designed to be a programming language, at least not in the sense we mean today. What we mean by a programming language is something we use to tell a computer what to do. McCarthy did eventually intend to develop a programming language in this sense, but the Lisp that we actually ended up with was based on something separate that he did as a theoretical exercise– an effort to define a more convenient alternative to the Turing Machine. As McCarthy said later,

Another way to show that Lisp was neater than Turing machines was to write a universal Lisp function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the Lisp function eval..., which computes the value of a Lisp expression.... Writing eval required inventing a notation representing Lisp functions as Lisp data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express Lisp programs in practice. What happened next was that, some time in late 1958, Steve Russell, one of McCarthy's grad students, looked at this definition of eval and realized that if he translated it into machine language, the result would be a Lisp interpreter.

This was a big surprise at the time. Here is what McCarthy said about it later in an interview:

Steve Russell said, look, why don't I program this eval..., and I said to him, ho, ho, you're confusing theory with practice, this eval is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into [IBM] 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today.... Suddenly, in a matter of weeks I think, McCarthy found his theoretical exercise transformed into an actual programming language– and a more powerful one than he had intended.

So the short explanation of why this 1950s language is not obsolete is that it was not technology but math, and math doesn't get stale. The right thing to compare Lisp to is not 1950s hardware, but, say, the Quicksort algorithm, which was discovered in 1960 and is still the fastest general-purpose sort.

There is one other language still surviving from the 1950s, Fortran, and it represents the opposite approach to language design. Lisp was a piece of theory that unexpectedly got turned into a programming language. Fortran was developed intentionally as a programming language, but what we would now consider a very low-level one.

Fortran I, the language that was developed in 1956, was a very different animal from present-day Fortran. Fortran I was pretty much assembly language with math. In some ways it was less powerful than more recent assembly languages; there were no subroutines, for example, only branches. Present-day Fortran is now arguably closer to Lisp than to Fortran I.

Lisp and Fortran were the trunks of two separate evolutionary trees, one rooted in math and one rooted in machine architecture. These two trees have been converging ever since. Lisp started out powerful, and over the next twenty years got fast. So-called mainstream languages started out fast, and over the next forty years gradually got more powerful, until now the most advanced of them are fairly close to Lisp. Close, but they are still missing a few things....

3 What Made Lisp Different

When it was first developed, Lisp embodied nine new ideas. Some of these we now take for granted, others are only seen in more advanced languages, and two are still unique to Lisp. The nine ideas are, in order of their adoption by the mainstream,

Conditionals A conditional is an if-then-else construct. We take these for granted now, but Fortran I didn't have them. It had only a conditional goto closely based on the underlying machine instruction.

A function type In Lisp, functions are a data type just like integers or strings. They have a literal representation, can be stored in variables, can be passed as arguments, and so on.

Recursion Lisp was the first programming language to support it.

Dynamic typing In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.

Garbage-collection

Programs composed of expressions Lisp programs are trees of expressions, each of which returns a value. This is in contrast to Fortran and most succeeding languages, which distinguish between expressions and statements.

It was natural to have this distinction in Fortran I because you could not nest statements. And so while you needed expressions for math to work, there was no point in making anything else return a value, because there could not be anything waiting for it.

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from Fortran into Algol and then to both their descendants.

A symbol type Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.

A notation for code using trees of symbols and constants.

The whole language there all the time There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

Running code at read-time lets users reprogram Lisp's syntax; running code at compile-time is the basis of macros; compiling at runtime is the basis of Lisp's use as an extension language in programs like Emacs; and reading at runtime enables programs to communicate using s-expressions, an idea recently reinvented as XML.

When Lisp first appeared, these ideas were far removed from ordinary programming practice, which was dictated largely by the hardware available in the late 1950s. Over time, the default language, embodied in a succession of popular languages, has gradually evolved toward Lisp. Ideas 1-5 are now widespread. Number 6 is starting to appear in the mainstream. Python has a form of 7, though there doesn't seem to be any syntax for it.

As for number 8, this may be the most interesting of the lot. Ideas 8 and 9 only became part of Lisp by accident, because Steve Russell implemented something McCarthy had never intended to be implemented. And yet these ideas turn out to be responsible for both Lisp's strange appearance and its most distinctive features. Lisp looks strange not so much because it has a strange syntax as because it has no syntax; you express programs directly in the parse trees that get built behind the scenes when other languages are parsed, and these trees are made of lists, which are Lisp data structures.

Expressing the language in its own data structures turns out to be a very powerful feature. Ideas 8 and 9 together mean that you can write programs that write programs. That may sound like a bizarre idea, but it's an everyday thing in Lisp. The most common way to do it is with something called a macro.

The term "macro" does not mean in Lisp what it means in other languages. A Lisp macro can be anything from an abbreviation to a compiler for a new language. If you want to really understand Lisp, or just expand your programming horizons, I would learn more about macros.

Macros (in the Lisp sense) are still, as far as I know, unique to Lisp. This is partly because in order to have macros you probably have to make your language look as strange as Lisp. It may also be because if you do add that final increment of power, you can no longer claim to have invented a new language, but only a new dialect of Lisp.

I mention this mostly as a joke, but it is quite true. If you define a language that has car, cdr, cons, quote, cond, atom, eq, and a notation for functions expressed as lists, then you can build all the rest of Lisp out of it. That is in fact the defining quality of Lisp: it was in order to make this so that McCarthy gave Lisp the shape it has.

4 Where Languages Matter

So suppose Lisp does represent a kind of limit that mainstream languages are approaching asymptotically– does that mean you should actually use it to write software? How much do you lose by using a less powerful language? Isn't it wiser, sometimes, not to be at the very edge of innovation? And isn't popularity to some extent its own justification? Isn't the pointy-haired boss right, for example, to want to use a language for which he can easily hire programmers?

There are, of course, projects where the choice of programming language doesn't matter much. As a rule, the more demanding the application, the more leverage you get from using a powerful language. But plenty of projects are not demanding at all. Most programming probably consists of writing little glue programs, and for little glue programs you can use any language that you're already familiar with and that has good libraries for whatever you need to do. If you just need to feed data from one Windows app to another, sure, use Visual Basic.

You can write little glue programs in Lisp too (I use it as a desktop calculator), but the biggest win for languages like Lisp is at the other end of the spectrum, where you need to write sophisticated programs to solve hard problems in the face of fierce competition. A good example is the airline fare search program that ITA Software licenses to Orbitz. These guys entered a market already dominated by two big, entrenched competitors, Travelocity and Expedia, and seem to have just humiliated them technologically.

The core of ITA's application is a 200,000 line Common Lisp program that searches many orders of magnitude more possibilities than their competitors, who apparently are still using mainframe-era programming techniques. (Though ITA is also in a sense using a mainframe-era programming language.) I have never seen any of ITA's code, but according to one of their top hackers they use a lot of macros, and I am not surprised to hear it.

5 Centripetal Forces

I'm not saying there is no cost to using uncommon technologies. The pointy-haired boss is not completely mistaken to worry about this. But because he doesn't understand the risks, he tends to magnify them.

I can think of three problems that could arise from using less common languages. Your programs might not work well with programs written in other languages. You might have fewer libraries at your disposal. And you might have trouble hiring programmers.

How much of a problem is each of these? The importance of the first varies depending on whether you have control over the whole system. If you're writing software that has to run on a remote user's machine on top of a buggy, closed operating system (I mention no names), there may be advantages to writing your application in the same language as the OS. But if you control the whole system and have the source code of all the parts, as ITA presumably does, you can use whatever languages you want. If any incompatibility arises, you can fix it yourself.

In server-based applications you can get away with using the most advanced technologies, and I think this is the main cause of what Jonathan Erickson calls the "programming language renaissance." This is why we even hear about new languages like Perl and Python. We're not hearing about these languages because people are using them to write Windows apps, but because people are using them on servers. And as software shifts off the desktop and onto servers (a future even Microsoft seems resigned to), there will be less and less pressure to use middle-of-the-road technologies.

As for libraries, their importance also depends on the application. For less demanding problems, the availability of libraries can outweigh the intrinsic power of the language. Where is the breakeven point? Hard to say exactly, but wherever it is, it is short of anything you'd be likely to call an application. If a company considers itself to be in the software business, and they're writing an application that will be one of their products, then it will probably involve several hackers and take at least six months to write. In a project of that size, powerful languages probably start to outweigh the convenience of pre-existing libraries.

The third worry of the pointy-haired boss, the difficulty of hiring programmers, I think is a red herring. How many hackers do you need to hire, after all? Surely by now we all know that software is best developed by teams of less than ten people. And you shouldn't have trouble hiring hackers on that scale for any language anyone has ever heard of. If you can't find ten Lisp hackers, then your company is probably based in the wrong city for developing software.

In fact, choosing a more powerful language probably decreases the size of the team you need, because (a) if you use a more powerful language you probably won't need as many hackers, and (b) hackers who work in more advanced languages are likely to be smarter.

I'm not saying that you won't get a lot of pressure to use what are perceived as "standard" technologies. At Viaweb (now Yahoo Store), we raised some eyebrows among VCs and potential acquirers by using Lisp. But we also raised eyebrows by using generic Intel boxes as servers instead of "industrial strength" servers like Suns, for using a then-obscure open-source Unix variant called FreeBSD instead of a real commercial OS like Windows NT, for ignoring a supposed e-commerce standard called SET that no one now even remembers, and so on.

You can't let the suits make technical decisions for you. Did it alarm some potential acquirers that we used Lisp? Some, slightly, but if we hadn't used Lisp, we wouldn't have been able to write the software that made them want to buy us. What seemed like an anomaly to them was in fact cause and effect.

If you start a startup, don't design your product to please VCs or potential acquirers. Design your product to please the users. If you win the users, everything else will follow. And if you don't, no one will care how comfortingly orthodox your technology choices were.

6 The Cost of Being Average

How much do you lose by using a less powerful language? There is actually some data out there about that.

The most convenient measure of power is probably code size. The point of high-level languages is to give you bigger abstractions– bigger bricks, as it were, so you don't need as many to build a wall of a given size. So the more powerful the language, the shorter the program (not simply in characters, of course, but in distinct elements).

How does a more powerful language enable you to write shorter programs? One technique you can use, if the language will let you, is something called bottom-up programming. Instead of simply writing your application in the base language, you build on top of the base language a language for writing programs like yours, then write your program in it. The combined code can be much shorter than if you had written your whole program in the base language– indeed, this is how most compression algorithms work. A bottom-up program should be easier to modify as well, because in many cases the language layer won't have to change at all.

Code size is important, because the time it takes to write a program depends mostly on its length. If your program would be three times as long in another language, it will take three times as long to write– and you can't get around this by hiring more people, because beyond a certain size new hires are actually a net lose. Fred Brooks described this phenomenon in his famous book The Mythical Man-Month, and everything I've seen has tended to confirm what he said.

So how much shorter are your programs if you write them in Lisp? Most of the numbers I've heard for Lisp versus C, for example, have been around 7-10x. But a recent article about ITA in New Architect magazine said that "one line of Lisp can replace 20 lines of C," and since this article was full of quotes from ITA's president, I assume they got this number from ITA. If so then we can put some faith in it; ITA's software includes a lot of C and C++ as well as Lisp, so they are speaking from experience.

My guess is that these multiples aren't even constant. I think they increase when you face harder problems and also when you have smarter programmers. A really good hacker can squeeze more out of better tools.

As one data point on the curve, at any rate, if you were to compete with ITA and chose to write your software in C, they would be able to develop software twenty times faster than you. If you spent a year on a new feature, they'd be able to duplicate it in less than three weeks. Whereas if they spent just three months developing something new, it would be five years before you had it too.

And you know what? That's the best-case scenario. When you talk about code-size ratios, you're implicitly assuming that you can actually write the program in the weaker language. But in fact there are limits on what programmers can do. If you're trying to solve a hard problem with a language that's too low-level, you reach a point where there is just too much to keep in your head at once.

So when I say it would take ITA's imaginary competitor five years to duplicate something ITA could write in Lisp in three months, I mean five years if nothing goes wrong. In fact, the way things work in most companies, any development project that would take five years is likely never to get finished at all.

I admit this is an extreme case. ITA's hackers seem to be unusually smart, and C is a pretty low-level language. But in a competitive market, even a differential of two or three to one would be enough to guarantee that you'd always be behind.

7 A Recipe

This is the kind of possibility that the pointy-haired boss doesn't even want to think about. And so most of them don't. Because, you know, when it comes down to it, the pointy-haired boss doesn't mind if his company gets their ass kicked, so long as no one can prove it's his fault. The safest plan for him personally is to stick close to the center of the herd.

Within large organizations, the phrase used to describe this approach is "industry best practice." Its purpose is to shield the pointy-haired boss from responsibility: if he chooses something that is "industry best practice," and the company loses, he can't be blamed. He didn't choose, the industry did.

I believe this term was originally used to describe accounting methods and so on. What it means, roughly, is don't do anything weird. And in accounting that's probably a good idea. The terms "cutting-edge" and "accounting" do not sound good together. But when you import this criterion into decisions about technology, you start to get the wrong answers.

Technology often should be cutting-edge. In programming languages, as Erann Gat has pointed out, what "industry best practice" actually gets you is not the best, but merely the average. When a decision causes you to develop software at a fraction of the rate of more aggressive competitors, "best practice" is a misnomer.

So here we have two pieces of information that I think are very valuable. In fact, I know it from my own experience. Number 1, languages vary in power. Number 2, most managers deliberately ignore this. Between them, these two facts are literally a recipe for making money. ITA is an example of this recipe in action. If you want to win in a software business, just take on the hardest problem you can find, use the most powerful language you can get, and wait for your competitors' pointy-haired bosses to revert to the mean.

8 Appendix: Power

As an illustration of what I mean about the relative power of programming languages, consider the following problem. We want to write a function that generates accumulators– a function that takes a number n, and returns a function that takes another number i and returns n incremented by i.

(That's incremented by, not plus. An accumulator has to accumulate.)

In Common Lisp this would be

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

and in Perl 5,

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

which has more elements than the Lisp version because you have to extract parameters manually in Perl.

In Smalltalk the code is slightly longer than in Lisp

```
foo: n
  |s|
  s := n.
  ^[:i| s := s+i. ]
```

because although in general lexical variables work, you can't do an assignment to a parameter, so you have to create a new variable s.

In Javascript the example is, again, slightly longer, because Javascript retains the distinction between statements and expressions, so you need explicit return statements to return values:

```
function foo(n) {
  return function (i) {
           return n += i } }
```

(To be fair, Perl also retains this distinction, but deals with it in typical Perl fashion by letting you omit returns.)

If you try to translate the Lisp/Perl/Smalltalk/Javascript code into Python you run into some limitations. Because Python doesn't fully support lexical variables, you have to create a data structure to hold the value of n. And although Python does have a function data type, there is no literal representation for one (unless the body is only a single expression) so you need to create a named function to return. This is what you end up with:

```
def foo(n):
  s = [n]
  def bar(i):
    s[0] += i
    return s[0]
  return bar
```

Python users might legitimately ask why they can't just write

```
def foo(n):
  return lambda i: return n += i
```

or even

```
def foo(n):
  lambda i: n += i
```

and my guess is that they probably will, one day. (But if they don't want to wait for Python to evolve the rest of the way into Lisp, they could always just...)

In OO languages, you can, to a limited extent, simulate a closure (a function that refers to variables defined in enclosing scopes) by defining a class with one method and a field to replace each variable from an enclosing scope. This makes the programmer do the kind of code analysis that would be done by the compiler in a language with full support for lexical scope, and it won't work if more than one function refers to the same variable, but it is enough in simple cases like this.

Python experts seem to agree that this is the preferred way to solve the problem in Python, writing either

```
def foo(n):
  class acc:
    def __init__(self, s):
        self.s = s
    def inc(self, i):
        self.s += i
        return self.s
  return acc(n).inc
```

or

```
class foo:
  def __init__(self, n):
      self.n = n
  def __call__(self, i):
      self.n += i
      return self.n
```

I include these because I wouldn't want Python advocates to say I was misrepresenting the language, but both seem to me more complex than the first version. You're doing the same thing, setting up a separate place to hold the accumulator; it's just a field in an object instead of the head of a list. And the use of these special, reserved field names, especially __call__, seems a bit of a hack.

In the rivalry between Perl and Python, the claim of the Python hackers seems to be that that Python is a more elegant alternative to Perl, but what this case shows is that power is the ultimate elegance: the Perl program is simpler (has fewer elements), even if the syntax is a bit uglier.

How about other languages? In the other languages mentioned in this talk– Fortran, C, C++, Java, and Visual Basic– it is not clear whether you can actually solve this problem. Ken Anderson says that the following code is about as close as you can get in Java:

```
public interface Inttoint {
  public int call(int i);
}
```

```
public static Inttoint foo(final int n) {
  return new Inttoint() {
    int s = n;
    public int call(int i) {
    s = s + i;
    return s;
    }};
}
```

This falls short of the spec because it only works for integers. After many email exchanges with Java hackers, I would say that writing a properly polymorphic version that behaves like the preceding examples is somewhere between damned awkward and impossible. If anyone wants to write one I'd be very curious to see it, but I personally have timed out.

It's not literally true that you can't solve this problem in other languages, of course. The fact that all these languages are Turing-equivalent means that, strictly speaking, you can write any program in any of them. So how would you do it? In the limit case, by writing a Lisp interpreter in the less powerful language.

That sounds like a joke, but it happens so often to varying degrees in large programming projects that there is a name for the phenomenon, Greenspun's Tenth Rule:

Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp. If you try to solve a hard problem, the question is not whether you will use a powerful enough language, but whether you will (a) use a powerful language, (b) write a de facto interpreter for one, or (c) yourself become a human compiler for one. We see this already begining to happen in the Python example, where we are in effect simulating the code that a compiler would generate to implement a lexical variable.

This practice is not only common, but institutionalized. For example, in the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of case (c), the human compiler, at work. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough– often that I'm generating by hand the expansions of some macro that I need to write.

Notes

The IBM 704 CPU was about the size of a refrigerator, but a lot heavier. The CPU weighed 3150 pounds, and the 4K of RAM was in a separate box weighing another 4000 pounds. The Sub-Zero 690, one of the largest household refrigerators, weighs 656 pounds.

Steve Russell also wrote the first (digital) computer game, Spacewar, in 1962.

If you want to trick a pointy-haired boss into letting you write software in Lisp, you could try telling him it's XML.

Here is the accumulator generator in other Lisp dialects:

```
Scheme: (define (foo n)
          (lambda (i) (set! n (+ n i)) n))
Goo:    (df foo (n) (op incf n _)))
Arc:    (def foo (n) [++ n _])
```

Erann Gat's sad tale about "industry best practice" at JPL inspired me to address this generally misapplied phrase.

Peter Norvig found that 16 of the 23 patterns in Design Patterns were "invisible or simpler" in Lisp.

Thanks to the many people who answered my questions about various languages and/or read drafts of this, including Ken Anderson, Trevor Blackwell, Erann Gat, Dan Giffin, Sarah Harlin, Jeremy Hylton, Robert Morris, Peter Norvig, Guy Steele, and Anton van Straaten. They bear no blame for any opinions expressed.

Related:

Many people have responded to this talk, so I have set up an additional page to deal with the issues they have raised: Re: Revenge of the Nerds.

It also set off an extensive and often useful discussion on the LL1 mailing list. See particularly the mail by Anton van Straaten on semantic compression.

Some of the mail on LL1 led me to try to go deeper into the subject of language power in Succinctness is Power.

A larger set of canonical implementations of the accumulator generator benchmark are collected together on their own page.

Japanese Translation.