After maybe 13 years of messing around with Lisp, on and off, I'm *finally* starting to like it.

Oh, sure, I've been lightly acquainted with Lisp's technical merits (at least some of them) for maybe a year now. I've been playing around with various Lisps and Schemes, and reading some books on them. And I've always done a moderate amount of Emacs-Lisp hacking. But until recently I never really *liked* Lisp. Even when I did my little Emacs typing-test application back in July, I only thought Lisp was so-so.

There were a few things I liked about it, sure. Most languages have a few areas where they really shine. Well, some do, anyway. But I've always felt like I was fighting with Lisp and Scheme, and when I really need to get something done, except in Emacs where you have no choice, I've always found it easier and more natural to work in Java or Perl or whatever.

I'm starting to like Lisp, though, and I'm finding I particularly like Scheme. It's quite different from elisp and Common Lisp. But I still think of them all as "Lisp", and I think that in spite of all the squabbling between Lispers and Schemers, they really *are* still Lisp.

In fact, fighting over them is like arguing the relative merits of, say, skiing vs. snowboarding. They're both a lot of fun. Skiers and boarders target each other for jokes, complaints, etc. That's natural, since they're the only ones on the slopes. They've got an entirely different set of problems than the kids on the inner tubes down by the parking lot.

The Functional Skiing Community

Actually, skiers and boarders aren't entirely alone on the slopes, just as Lispers and Schemers aren't entirely alone in their problem space. There's the occasional Haskell-lover on snowblades, or an OCaml person on telemark skis. Lispers and Schemers consider them part of the family — it's hard not to when they're there on the same slopes, going off the same jumps with you, looking weird but competent. But they're a fairly small minority.

Is it just me, or do those telemark skiers always look like they're world-class? I'm talking about those long skis where the heels lift way out of the bindings, so they have an entirely different style than Alpine/downhill skiiers. They always seem to be coming down out of the forest above the runs I'm on. Maybe you have to be world-class if you want to master something not many others are doing.

They also look pretty lonely. Snowboarders clump together with other boarders, skiers with skiers, and so on. This is partly just (sub-)cultural — boarders like the baggy clothes and all that — but also partly because the natural arcs skiers and boarders make coming down the mountain are different, so they have an unfortunate tendency to collide at high speeds. Telemarkers' tracks are different from both. So telemarkers, at least in the ski resorts I've frequented, seem to be out on their own most of the time. But they certainly know their stuff.

I remember it took me a few years before I really *loved* snowboarding. It didn't happen until I was fairly advanced, and could go down any "sane" slope (i.e. some double-blacks — no sharp rocks or 20-foot sheer drops for me, thanks) without fear. I thought I loved boarding early on, but each year it kept getting better, as I got better.

I never got past intermediate level as a skier. I only did it for 2 or 3 seasons, and always used rental skis, so I wasn't comfortable on black-diamond slopes. I've been boarding for 12 or 13 seasons now, and after taking some advanced lessons, my technique got dramatically better. You never really notice how much energy you're wasting by doing things inefficiently, at least until someone actually shows you. That's true of boarding, and programming, and probably just about everything else as well.

I've sometimes wondered how master-level techniques are discovered for totally new skill domains.

Maybe some people are just naturally good at it, and a few are articulate enough to show other people what they're doing. Maybe some people have skills that carry over from other domains, so initially the best boarders might have been surfers or skiers or skateboarders. And maybe some people have just figured out that there are universal concepts like efficiency and economy of motion that apply to mastering just about anything. Who knows.

Obviously the experts and masters of any skill have a much richer, deeper appreciation for it than a beginner. I suspect that in general, that deep appreciation allows them to derive more enjoyment from their art/sport/craft than people who are less skilled in it. I don't know that for sure, but it's what all the circumstantial evidence seems to say. Clearly non-masters can have a lot of fun too. But most things seem to become more enjoyable as you get better at them, and spend less time fighting with insufficient knowledge or bad technique. You can focus that much more on doing cool stuff.

If your art or sport or whatever is demanding enough, so that even the masters can continue to improve, then it rarely loses its novelty. You can have fun sliding down a slope on an inner tube, and I suppose some people can enjoy the experience for their whole lives. But when you learn how to ski (and it's a lot of painful work!), new vistas open up that you probably weren't expecting. To kids on inner tubes, it looks like skiers are just going up and down crowded slopes, so it's hard to imagine that it's any more fun than tubing and building snowmen.

Even most skiers and boarders never become skilled enough to experience the breathtaking beauty and solitude of the back-country. Or the profound satisfaction of controlling your path through moguls, jumps, trees, and high-speed chutes. But it just keeps getting more fun, and it's clear that the boarders better than me are loving it even more than I am. Struggling takes a lot of the fun out of things.

Late-Night Scheming

Lisp and Scheme basically snuck up on me. Until *very* recently, even though I've been studying and using them both quite a bit, I still preferred Java and Ruby, and even C++ or Perl, over Lisp for real work. I just hadn't discovered that back-country wilderness yet. Part of the key to understanding Lisp is realizing that you won't ever see its strengths if you just try to write your usual stuff in it. Down on the beginner slopes, telemark skis and world-class artistry are fairly useless, because all you can really do is slide along at 10 mph. You don't need any real sophistication until you try tackling the the mountaintops.

I stayed up late last night doing programming exercises in Scheme. That's certainly not what I'd planned on doing. But finally, after months of reading (books, websites, source code) and experimentation, something "clicked". All of the sudden I was obsessed with it. It was almost like I was discovering programming for the first time.

That didn't happen with, say, Ruby. It only took a few days to learn a substantial amount of Ruby, and I immediately felt happier about solving the problems I used to solve with Perl. Same with Java: it was just a better C++, and Perl had been a better awk/sed/sh, etc. I got very excited about all of them, and I still am, to some extent — you can't help but love all languages a little, after doing 5 years of nothing but hardcore assembly-language programming. But learning those languages didn't feel like rediscovering programming.

I'll confess readily: Lisp me took a long time to learn. I've screwed around with Lisp for *years* without liking it much — at least not enough to want to do any day-to-day programming in it. And even after embarking on a serious undertaking to become proficient with it, it still took me more than a year of reading and practicing before I finally started to really like it. And then it happened more or less all at once. In that sense, it really *was* like learning my first language.

One big difference may have been my switching to focus more on Scheme, recently, but it's hard to say. Maybe it just takes a long time. Or maybe you don't actually try as hard to learn things until you really believe, deep down, that it will be valuable. I do distinctly remember when I decided that I really was going to use Lisp or Scheme for something major, the books got a *lot* more interesting. It was a weird, instant transformation from "academically interesting" to "I want to learn this stuff right now."

Even after I'd committed to really learning it, Lisp still felt, compared to languages like Java and Ruby, a bit like trying to go from tubing to telemarking. It wasn't intuitive, it didn't make me feel comfortable or happy, and it didn't seem to offer much value over other languages.

Anyway, once I *finally* got excited about it, which I'd put right around "yesterday", I started making my way through the exercises in various textbooks, even doing those 3-star ones at the chapter ends that nobody ever does. I also made up my own challenges as I went. I spent an hour writing three or four versions of quicksort, and I can assure you I've never before had the urge to implement quicksort for the "fun" of it.

Another exercise from one of the books was to walk a code tree and compute the lexical-address info for all the identifiers, annotate the tree with it, and then reconstruct the variable name from its address later. Um. I typically like to write web sites and games and scripts and stuff — not pieces of compiler back-ends. But last night I wrote countless tree-walkers, mini theorem provers, all kinds of things that I'd heard of but never implemented, except maybe for specific course projects back in school, after which I promptly and eagerly forgot everything about them.

Last night, in one night, I did stuff that probably would have taken a week or two in Java, a language I'm vastly more familiar with. I'll never know for sure, though, because I'd never want to implement *any* of those things in Java. They just wouldn't be fun. I tried doing a few of them in Ruby, and it wasn't too bad, but it definitely wasn't "fun". The same algorithm, right after I'd done it in Lisp, felt like work in Ruby. Evidently the problems that Lisp is immediately suited for, right out of the box, are problems that are fairly painful in Java or C++, and none too fun in Perl, Ruby or Python either.

That's one realization, but it's far from the full picture. What's also become clear is that Lisp is better at *adapting* to new problem domains than any other language I've used. And I haven't used other functional languages like Haskell and OCaml much, but I suspect Lisp beats them in adaptability as well.

Evolution is King

Perl began as a scripting language, but it quickly adapted itself (via libraries and language extensions) to become a good language for Web programming as well. Perl was really the first flag planted on that new moon, most likely because of its strong text-processing capabilities and Unix integration. But Perl tends not to be used as much for the Web these days; it's nowhere near as popular for building websites as PHP, a Perl-like language made specifically for the web. They probably wouldn't have felt the need to create PHP if Perl had been sufficiently adaptable.

As another example, Perl has never made significant inroads into the embedded-language space. Perl ought to be an excellent language for embedding in other applications; i.e. an " app scripting" language, which is after all very similar to Perl's original purpose. But it turns out to be a lousy embedded language for lots of reasons, most of them boiling down to Perl's lack of adaptability, which most people don't notice because it's got so many shortcuts for Unix scripting and Web programming. So cleaner languages like Python, Tcl, Lua and even Visual Basic now have the lion's share of the embedded-language space.

Java has proven itself to be a moderately adaptable language — either that, or it's just had a lot of

people muscling it forward. It began life intended as a replacement for C++ on embedded devices, detoured as a rich web-client language, and wound up taking a good deal of the enterprise-server space (and other domains). And it's well on its way towards achieving its original goal of being the dominant application platform for mobile devices. Java must be doing something right.

C is even more adaptable than Java. It's everywhere, and you can do anything with it, although it really starts to break down at a certain size system — perhaps a million lines at most. Java scales to much larger code bases, although it unfortunately eats away much of this advantage by being inescapably verbose. Like my blog!

C++ is much less adaptable than C. It's large, nonstandard, ungainly, and nonportable, and it has horrible linking problems, regrettable name-mangling, a template system that's too complex for what you can do with it, and so on, and on. These things all hurt its ability to evolve towards suitability for new (or specialized) problem domains. C++ was able to move into the application-programming domain during a temporary historical anomaly, from the mid-80's to mid-90's, during which software demands outpaced Moore's Law for a while. The balance is restored, but C++ is still hanging around like a crusty old aunt, because of all the legacy app code out there. How C++ made it onto the server-side of the web, I'll never know. It's hit the Peter Principle for languages big time: promoted far beyond its capabilities.

Designing for growth

What makes a language adaptable? Well, languages are how you say things, so if you need to start expressing new concepts, your language needs to be able to grow to express them. In particular, you have to be able to express them *cleanly*, without having to jump through strange hoops, mix in a bunch of historical gunk, and write tons of comments that say: "warning: severe hack ahead". So that rules out Perl and C++.

A language also needs to be well-specified. If nobody's exactly sure how the language is supposed to work, then it's difficult for people to build tools for it, develop implementations on new platforms, and build a community around it. Having a rigorous language spec was a huge part of Java's success. C++'s spec is very large, but is still filled with holes. Perl's spec is a printout of Larry's source code, which looks the same in ascii, ebcdic, and gzipped binary form.

It's easiest to make a solid language specification if the language is small, orthogonal, and consistent. Again, this rules out both C++ and Perl. If the language has highly complex syntax and semantics, then you've got a lot of writing ahead of you, and your spec may still never be very solid. Java has only moderately complicated syntax and semantics, and the language specification was created by some of the world's leading Lisp experts (Gosling, Joy, Steele, and others), so they were able to do a first-rate job.

In addition to relative simplicity (at least syntactic simplicity) and a thorough spec, another key element of adaptability is that the language needs to be extensible. If a particular feature of the language is getting in your way in some new problem domain, then you need to be able to change it, or soon you'll be looking around for a new language.

C provides a simple but reasonably effective extension mechanism via its preprocessor macros. These proved very useful for a wide variety of problems not addressed by the language, such as feature-conditional compilation. In the long term, though, it doesn't scale very well, being neither very expressive nor very well-integrated with the language and tools.

But at least C *has* a macro system. Java has absolutely nothing in this space — admittedly the C preprocessor wasn't something you'd want to copy, and C++ templates are just about the ugliest thing

ever invented by the hand of man, but that doesn't mean you should leave the problem entirely unsolved. Now there are all kinds of Java preprocessors, code generators, huge frameworks like AspectJ, new languages for the JVM and so on, sprouting up everywhere, all designed to give you the ability to make Java flex a little.

The lack of a macro system may well be the thing that finally kills Java, in ten years or however long it's got left. Don't get me wrong: I think Java's a very strong platform, with lots going for it, and Java introduced (or at least packaged together) a lot of novel ideas that people take for granted now. But a language has to bend or break, and Java's not doing much bending. Very little of the innovation in Java these days is with the language, and almost none of it has fundamentally improved its expressive power or its extensibility. Java's turning out to be the New Pascal.

Python's a very adaptable language, and does well in just about all of the categories above except for macros. It does offer metaprogramming, which lets you solve similar classes of problems, albeit not with any assistance from the compiler. So it's found lots of niches and has a pretty good user following. Good examples of its adaptability include Jython (a port of Python to the Java VM) and Stackless Python, which uses continuation-passing for its control flow. Python's also frequently used as an embedded scripting language. All in all, it's a pretty darn good language. Google uses it a lot, or so I've heard.

Unfortunately Python has some flaws that may prove fatal, and which have undoubtedly kept it from being more widely adopted. One is that the user community consists mostly of frigid, distrustful, arrogant, unfriendly jerks. Wasn't it Thomas Hobbes who first observed that communities and organizations tend to reflect the personalities of their leaders? Python has other problems, such as the lack of optional static types, although they're talking a lot about adding that in. But most of Python's technical problems would be easily surmountable if they weren't such a bunch of kneebiters.

I could go on comparing other languages, but you get the idea. Times change, and new problems arise, but languages generally don't evolve very quickly: they always have backward-compatibility issues, ones that grow more onerous as time goes by. The only way to keep up is to provide users with language-extensibility mechanisms.

Of course, defining functions and types is the main way people grow their languages to fit their problem domains. Those two mechanisms (functions and an extensible type system) can carry you a very long way. The Java community has gotten pretty far with Java's simple class system, mostly through sheer willpower.

But for Java it's beginning to break down. People are using code generators (EJB, WSDL, IDL, etc.), and they're offloading processing that should logically have been in Java into XML, e.g. Ant and Jelly, to cite but two examples. Java has sprouted a powerful 3rd-party metaprogramming system called AspectJ, and it's on the verge of becoming accepted by the majority, which means Sun is losing control of the language. Many other languages are appearing for the JVM, not to mention templating preprocessors like JSP. Java interfaces and code bases are becoming so huge that they have turned towards "refactoring tools" as a way to help them move masses of similar-looking code around.

If Java's so fugging cool, then why does all this stuff exist?

That's not a rhetorical question. I really want to know.

Some people claim that Java is just naturally the best language for solving large-scale business problems, so Java happens to be tackling the biggest problems in the world, and it's hitting scaling walls long before other languages will see them.

That's not what I think. My theory is that Java naturally makes things large and complicated. So large

and complicated, in fact, that it requires unusually sophisticated automation tools just to keep normal-sized projects under control.

Take a look at any large Java framework these days (they only come in one size) and you'll see what see what I mean. JMX is a good random example, since it just got added to Java 5. It's hundreds of classes, thousands of interface methods, and several large-ish mini-languages passed around in String objects. For what, you ask? Well, it provides a mechanism for you to monitor your app remotely. Er, sort of. It actually provides a mechanism for you to *build* remote monitors. I'd have thought that would be a relatively straightforward problem, and more importantly one that's already been solved in various ways. But JMX is all-new, and all-huge, and it's not even generic or reusable, at least according to its documentation.

Actually, I'll get sidetracked if I spend too long on Java frameworks. I'll save it for another blog someday, maybe. Besides, I don't want to come down too hard on Java, because it has a LOT of things going for it that it does really well. And, ironically, AspectJ may well turn out to be the thing that keeps Java in the game. I had no idea just how powerful or mature it was until Gregor came and gave his talk, and even then his talk didn't do it justice. I just happened to be assigned as his "buddy" from Dev Services, so I grilled him for half an hour before the talk, and again for half an hour afterwards, and listened to other people grilling him as well. I'm convinced now, big time, but even so, it's going to take a lot of slow, careful experimentation before I feel really comfortable about its suitability for production work.

And besides, Java's not the only language that has limitations. C++ broke down a loooong time ago. It's just a body shop now. The standard quantity of measurement for C++ tasks is the "body", and the units typically range from dozens to hundreds.

Complexity Boundaries

The reality is that every language has a natural, built-in limit to its ability to help humans manage complexity before it poops out. Every language has a natural maximum size for systems that can be written in that language. Up to that maximum size, things go really well, and you spend a lot of time focusing on solving the business problem.

As the system grows beyond the language's natural complexity boundary, people start to spend more and more time wrestling with problems caused by the language itself. The code base gets huge, and the perpetual bug count grows linearly (at least) with the size of the code. All kinds of inertia sets in, as people begin to realize that making certain changes will require modifying thousands of files, potentially not in an automated way, because tools can generally help you with syntactic problems, but not so much with semantic or architectural ones. It starts to become overwhelming, and eventually progress slows to a trickle.

This has happened to me *twice* now outside of Amazon. The first time was at Geoworks. We wrote everything by hand in assembly language: apps, libraries, drivers, kernel — an entire desktop (and eventually handheld) operating system and application suite. The system eventually collapsed under its own weight. Many Geoworks people still don't like to admit it, but that's what happened. You can laugh it up, saying how foolish we were to write everything in assembly language for performance reasons.

But you're doing it right now at Amazon. Same justification, same approach, and already some of the same outcomes. C++ *is* assembly language: low-level, non-portable, occupying no useful niche; squeezed out on the low end by straight C, squeezed on the high end by Java and even higher-level languages that are now shining as Moore's Law runs ahead.

I know, I know: you and lots of smart people you know disagree. That's fine. You're welcome to

disagree. I can't blame you: I and everyone else at Geoworks felt the same way about assembly language. We were doing great things with it. C was for wimps, and C++ was for Visual Basic pansies. Performance was King. We all have a blind spot for performance, you know. All programmers have it. It's in our bones, and we're very superstitious about it. So I hear you, and I'm not going to try to change your mind. Everyone has to figure it out for themselves.

I didn't figure it out until it had happened to me *again*, seven years later, after building a very large Java application mostly by myself. At some indefinable point, the bulk of my effort had shifted from extending the application to maintaining it. Never satisfied, I did several months of deep analysis on the code, and finally concluded that many/most of the problems were intrinsic to Java, hence unavoidable. I had been immersed for seven years in blind devotion to Java, seeing as it *was* actually kinda nice compared to the old assembly-language gig, and my findings were surprising and frustrating.

So I spent another year or so of my free time on a massive quest, searching for a Java replacement. It wasn't as easy as I'd hoped. Lots of promise, lots of potential, but precious few high-level languages that actually deliver the vast number of tools and features you need for building production systens. Given that AOP can help with about half the problems I was having, Java is still a very solid choice. There's really only one other contender, one I probably should have been using from the beginning.

But I won't bore you with the details. The point I was trying to make in this section is that all languages have a natural system-size limit, beyond which your innovation starts to slow down dramatically. It happens regardless of how well you've engineered your system, and regardless of whether your team is one person or a thousand. Overall productivity never goes to zero, but that's moot; the point is that you started out strong and now you're weak.

The natural complexity boundary for any given language is also a function of your engineering quality, but that can only reduce the upper bound. The natural limit I'm talking about is the one you hit even if your system is engineered almost perfectly. Adding engineers can help up to a point, but eventually communication, code-sharing, and other people-related issues outweigh the benefits.

You might think your problem domain determines the upper complexity bound more than the programming language you're using. I don't think so, though, because eventually *all* systems grow to the point where they're huge, sprawling, distributed/networked, multi-functional platforms. Back in the 70's and 80's, Unix people joked that all applications eventually grow to the point where they can read mail, and any that don't are replaced by ones that can. There was some truth to that. Today, systems grow to the point where they speak Web Services and other protocols, or they're replaced by ones that can. I don't think your problem domain matters anywhere near as much as the language, because in some sense we're all converging towards the same kinds of systems.

What language-level features contribute to a language's scalability? What makes one language capable of growing large systems, while a similar system in another language spends all its time languishing in maintenance-mode?

Type Systems

If my 18 months of intense study have answered any question at all, it's what data types are exactly, how they work, and why I'm always struggling with whether I prefer strong or weak typing.

The design of the static type system is an important determinant of language scalability. Static type annotations improve system safety (in the quality sense, if not necessarily in the security sense). They also help the compiler produce optimized code. They can help improve program readability if you don't overdo them. And type tags help your editor/IDE and other tools figure out the structure of your code, since type tags are essentially metadata.

So languages without at least the *option* of declaring (or at least automatically inferring) static types tend to have lower complexity ceilings, even in the presense of good tools and extensive unit testing.

You might think C++ and Java are strongly typed. You'd be wrong. Ada and ML are strongly typed. C++ and Java have some static typing, but they both provide you with lots of ways to bypass the type system. To be honest, though, I think that's good thing; type-checker algorithms are often not very smart, and sometimes you need them to get out of your way. In the end, if your strongly-typed system becomes too much of a pain, you'll find ways around it — e.g. passing XML parameters as strings through your IDL interfaces, or whatever.

It might seem odd that type systems yield such great benefits, yet when you make them too strict, nobody can stand them anymore. The problem is deeper than you might suspect, particularly if you subscribe to the view that Object-Oriented Programming is the be-all, end-all of type systems. Take another look at that JMX interface. It's strongly typed — superficially, at any rate. But if you dig one or two levels deeper, you'll see they gave up on the rigorous typing and highly-specialized 32-letter identifier names. At the lowest level, it's all weakly-typed string parameters and oddly generic-sounding classes like "Query" and "Role". Almost as if there's a query language frozen in that mass of OOP interfaces, like Han Solo's face sticking out of the carbonite.

The whole interface is massively overcomplicated, and it seems to exhibit a sort of struggle between strong and weak static typing, right there in the same framework. What's going on? Is this really so hard?

The problem is that types really are just a classification scheme to help us (and our tools) understand our semantic intent. They don't actually *change* the semantics; they just *describe* or *explain* the (intended) semantics. You can add type information to anything; in fact you can tag things until you're blue in the face. You can continue adding fine-grained distinctions to your code and data almost indefinitely, specializing on anything you feel like: time of day, what kind of people will be calling your APIs, anything you want.

The hard part is knowing when to stop. If you're struggling over whether to call a method getCustomersWithRedHairAndBlueNoses or getCustomersByHairAndNoseType(HairType.RED, NoseColor.BLUE), then you've officially entered barkingUpWrongTreeTerritory. Over-zealous typing is a dead giveaway that someone junior is doing the design. At the other exteme, not putting in enough static type information is certain to create problems for you, as you muddle through your nests of untyped lists and hashes, trying to discern the overall structure.

This is the central problem of Object-Oriented design, and of program and system design in general.

And if designing your own types is hard, getting a language's type *system* right is much harder, because there really is no "right" — it's different for different people. Some people clean their homes until every last speck of dust is gone, and some people are a bit sloppier, but we all manage to get by. It's mostly a matter of personal preference. This is why advocates of "weak" typing don't see any advantage to Java's interface declarations and fine-grained primitive types. They're going to unit-test it all anyway; the type system is just redundant bureaucracy to them.

Why do you need ints and longs, for instance? A good system should automatically use the smallest representation it needs, and grow it on demand, rather than overflowing and introducing horrible bugs, as Java and C++ both do. Having fine-grained types for numeric precision just gets people into trouble. In the real world, numbers have distinct, naturally-occurring types that don't really map to the primitives you find in popular languages, most of which chose their built-in types based on machine word-sizes. In nature, we have whole numbers, natural numbers, integers, rational numbers, real numbers, complex numbers, and then an infinite chain of matrices of increasing dimensionality. That's

just how it works, and you sort of want your programming language to work that way too, if you think about it.

Java's Type System

Many Java advocates are very excited by the notion of automatically-generated API documentation. Gosh, I guess I am too. But because JavaDoc does such a great job, and we rely on it so heavily, many people tend to confuse Java's type system with the doc-generation system, and they think that you can't have that kind of documentation without being able to declare Java-style interfaces.

It's simply not true. Java definitely raised the bar on auto-documentation. But Javadoc's output quality isn't better because of the interface signatures; the quality is almost entirely due to the JavaDoc tags, which you fill in by hand. The documentation written by humans is far more useful than the dubious declaration that a function takes two ints, or two Strings, or even two Person objects. You still need to know what it *does* with them, and whether there are any constraints on the data that aren't expressed by the type signatures — e.g. whether can you pass in the full range of 32-bit integers to the function.

But many Java enthusiasts have latched onto interfaces, and they appear to think of interfaces as the holy grail of type systems. They know, deep down, that Java's type system is non-orthogonal, inflexible, and not very expressive, and they ask questions about how to get around problems with it on a daily basis. But they're used to assuming it's how the universe works, so you don't question it, any more than you question why you have to commute to work when you ought to be able to teleport there instantly. (The physical universe's constraint system sucks, too.)

It would take a whole book to explain all the problems with Java's type system, but I'll try to throw out a few examples.

One obvious problem is that Java's type extension mechanism is limited to defining classes and interfaces. You can't create new primitive types, and there are severe limitations on what you can do with classes and interfaces. As a random example, you can't subclass a static method — a feature that would occasionally be extremely useful, and which is present in other languages. And a class can't get its own name at compile-time, e.g. to pass to a static logger. (Jeez.)

One that really irks me: there's no such thing as a "static interface" — an interface full of static methods that a Class object promises to implement. So you have to use reflection to look up factory methods, hardwire class names in your code, and so on. The whole situation is actually quite awful.

As a result of Java's whimsical limitations, you often find objects or situations in the real world that are very difficult to express in Java's type system (or C++'s, for that matter, on which Java's is modeled). Many Design Patterns are in fact elaborate mechanisms for working around very serious limitations of the C++ and Java type systems.

An even more subtle point is that every single element of a programming language has a type, even though the language doesn't actually recognize them as distinct entities, let alone typed ones.

Here's an example of what I mean by being able to tag things until you're blue in the face: Take a for-loop. It's got a type, of course: for-loop is a type of loop. It can also be viewed as a type of language construct that introduces a new scope. And it's a type that can fork program control-flow, and also a type that's introduced by a keyword and has at least two auxiliary keywords (break and continue).

You could even think of a for-loop as a type that has lots of types, as opposed to a construct like the break statement, which doesn't exhibit as much interesting and potentially type-worthy behavior. A type is just a description of something's properties and/or behavior, so you can really get carried away with overengineering your type declarations. In extreme cases, you can wind up with a separate Java

interface for nearly every different method on a class. It's pretty clear why Python and Ruby are moving towards "duck typing", where you simply ask an object at runtime whether it responds to a particular message. If the object says yes, then voila — it's the right type. Case closed.

Ironically, even though every single Java language construct has many (possible) types, Java itself is *oblivious* to them. Even Java's runtime reflection system is only capable of expressing the types provided by the OO mechanism (plus some stragglers like the primitive types). Reflection has no visibility inside of methods. If you want to write Java code that reads or writes Java code, then you have to come up with your own object model first, and potentially wrestle with complex parser generators and so on.

Many languages that don't offer you much in the way of static type annotations (e.g. Ruby, Python, Perl) still have tremendously rich type systems. They provide you with more flexibility in defining your types, using far less code, and still providing the same level of safety and automatic documentation. Declaring a bunch of types doesn't make your code safe, and not declaring them doesn't make it unsafe. Types are just helpers, and making the right type choices is a fuzzy art, one that boils down to taste.

This is one reason OO design (and, by extension, service interface design) is so difficult, and why interview candidates are often so "bad" at it. I'll have more to say about types in a future essay, I'm sure.

I went slightly off-course in this section. My goal was to illustrate that there's life beyond Java, that Perl programmers aren't *quite* as disorganized as most Java folks would like to believe, and of course that a language's type system is one of the most important contributors to how well the language "scales".

Language Scalability

I could list other factors that contribute to language scalability: the complexity of the syntax, for instance, is a real barrier to scaling, for many reasons. You really don't want a complicated syntax; it'll come back to bite you. Perl, C++ and Java all have artificially complicated syntax: too much for too little semantic benefit. Java is the least bitten of the three, but it still inherits a lot of cruft from C++, which was designed by a rank amateur. (As was Perl.)

The module system is another big one. Runtime introspection support is another. In fact, all of the choices made in designing a language, whether explicit or inadvertent, have some impact on how far the language will scale before your system becomes too complex for human beings to keep evolving it.

But all of these contributors pale in comparison to *extensibility* — i.e., the extent to which users can customize or change the behavior of the language itself to suit their needs.

And *all* languages, bar none, pale in comparison to Lisp when it comes to extensibility (or adaptability, as I was calling this property earlier). It doesn't matter how cool or sophisticated or far-reaching you think your favorite language's extension mechanisms are: compared to Lisp, all other languages are orders of magnitude less capable of evolution.

Lisp is DNA

Alan Kay (co-inventor of Smalltalk and OOP) said of Lisp: "it's not a language, it's a building material." Some authors have called it a "programmable programming language". Some people say: "it looks like fingernail clippings in oatmeal." People have in fact said all sorts of amusing and interesting things about Lisp.

All of the descriptions fall short of the mark, though. And mine very likely will, too. But I'll try. Or at least I'll try to give you the barest outline in the next few paragraphs.

Every programming language runs on a *machine*. That machine is NOT the hardware, because all languages can be made to run on different machines — even assembly language, which can be run on emulators. Programming languages are built atop an abstract machine conceived by the language designer. This machine that may or may not be very well specified. For instance, many languages have at least a few constructs whose behavior relies on the implementation of a system-dependent system call.

All languages have at least part of their underlying machine's functionality implemented in software. Even C and C++ have software runtime environments, and of course both languages also rely heavily on OS services, which are also implemented in software and supported by the hardware. In fact, all programs run on a tower of machines. Even the hardware, which you might think of as pretty "low level", is actually constructed out of smaller machines, all the way down to the level of quantum mechanics.

OK. Got it. Programs in all languages run on virtual machines.

Languages also provide some amount of syntax: a set of input symbols you can use for constructing your program, a set of mostly-formal rules governing how you may arrange those symbols, and a set of mostly-informal descriptions of how your symbol arrangements will be interpreted.

Most languages limit you to the symbols in the ASCII character set, for historical reasons, and they're gradually migrating toward Unicode. But a symbol could be anything at all, as long as it can be distinguished from all other symbols used by the language. It's just a unique string of bits, usually with a predetermined way of displaying and/or entering it.

The machine interprets your program's symbols according to the language rules. Hence, all machines are interpreters. Some of the interpreting goes on in hardware, and some in software. The hardware/software boundary is flexible. For instance, CPUs have started offering floating-point arithmetic in hardware, and video cards now offer polygon rendering and other algorithms that used to be pure software.

With me so far?

Compilers are just programs that pre-interpret as much of your program as possible. Technically they're unnecessary; you can interpret any language directly from the input symbols. Compilers are a performance optimization. They're quite useful, and we grossly under-utilize them, but talking about performance would take me a bit too far afield today. I'll have more to say about performance at some point, I'm sure.

Hardware is also a performance optimization, if you separate the notions of storage and computation. Ultimately computations can be performed by people or any other kind of physical process that knows how to interpret your program. When you hand-simulate your program, you're the machine.

So "interpreters" are a much more fundamental notion than hardware or compilers. An interpreter is a tower of machines, some software and some hardware, running at different times, all working together to execute your program.

And your program is just a bunch of instructions, specified by your arrangement of the language's symbols according to the rules of the language. Languages and interpreters go hand in hand.

Some programming languages are designed with the goal of being interpreted directly by the hardware, with as little intervention from software as possible. C is such a language. This approach has the

advantage of being pretty fast on current hardware. It has all the disadvantages attendant to premature optimization, but I can't get started on performance; I've already deleted about ten paragraphs about it. I'll leave it for another blog.

Some languages, e.g. C++ and Java, are designed to run on more abstract machines. C++'s abstract machine is a superset of the C machine, and includes some OOP abstractions. C++ has very complex syntax, partly due to inexperienced design, and partly because it made many concessions to hardware performance over people performance. Java runs on a virtual machine, but it was designed to map very closely to existing hardware, so in reality it's not so far removed from the bare metal as people tend to think.

Perl has its own complex, baroque, ad-hoc machine: the Perl interpreter. Perl has a very complex syntax, with many shortcuts for common operations.

When you create classes and functions and libraries, you're not extending the programming language. You're extending the machine. The language stays the same. If it was hard to say certain things before, then adding libraries of classes and functions doesn't really help.

For instance, in Java you must double-escape all the metacharacters in your regular expressions, not to mention create Pattern and Matcher objects, all because you can't make any changes to Java's syntax. Unless you want to use the hundreds of non-standard preprocessors that have sprouted up precisely for reasons like this one.

Another example: you can't write logging, tracing, or debugging statements intelligently in Java. For instance, if you want to write something like this:

```
debug("An error occurred with this object: " + obj);
```

where "debug" is a function that checks whether a debugging flag is set, and if so, prints the message somewhere.

If you do this, someone will happily point out that in Java, function arguments are evaluated *before* calling the function, so even if debugging is turned off, you're doing a bunch of work with that string concatenation: creating a StringBuffer object, copying in the string literal, calling a polymorphic toString() function on obj, creating a new string to hold the object's description, which possibly involves more StringBuffers and concatenation, then returning the StringBuffer on the stack, where its contents are appended to the *first* StringBuffer, possibly resulting in a reallocation of the memory to make room, and then you're passing that argument on the stack to the debug() function, only to find that debugging is off.

Oops.

And that's the best-case scenario. In the worst case, you could trigger a thread context switch, or a garbage collection, or an unneeded operating system page-cache fetch, or any number of other things that you really don't want happening in production because of that debug line, at least when debugging is off.

There's no way to fix this. Even a preprocessor couldn't do it for you. The standard hacky workaround is to write something like this instead:

```
if (DEBUG) debug("An error occurred with this object: " + obj);
```

where "DEBUG" is a constant in your code somewhere. This approach is fraught with problems. The debug flag may need to be shared across multiple classes, so you need to either declare it in each class, or export it from one of them. The name of the flag appears right there, inlined with your code, and

there's no way to avoid typing it in hundreds of places (unless you use AspectJ). And it's just plain ugly.

My third and final Java example: sometimes you really do need multiple inheritance. If you make a game, and you have a LightSource interface and a Weapon interface, and behind each interface is a large implementation class, then in Java you're screwed if you want to make a Glowing Sword. You have no recourse but to manually instantiate weapon and light-source implementation objects, store them in your instance data, implement both interfaces, manually stub out every single call to delegate to the appropriate instance, and hope the interface doesn't change very often. And *even then*, you haven't fully solved the problem, because the language inheritance rules may not work properly if someone subclasses your GlowingSword.

The regexp-escaping problem is a lexical problem: an eval-time or compile-time macro system won't help you, because the lexical analyzer has already done its dirty work before the parser ever sees the string. If you wanted to provide a way around this in Java, without using a preprocessor (which is a hack), you'd need an API that allows you to interact with the lexer. That's all. Just an API, and a way to arrange to invoke it before the rest of your code is lexed.

The debug-flag problem is an evaluation-time problem. You can fix problems like this either by adding support for lazy evaluation to your language, or by adding a macro system. They amount to mostly the same thing, except a macro system lets you add some syntactic sugar as well, which is sometimes appropriate.

The multiple-inheritance/delegation problem is a problem with the interpreter semantics not being flexible enough. It manifests later than eval-time, and it's conceivable that you could fix it without needing to change the language syntax. For instance, if Java simply had a methodMissing method in java.lang.Object, one that was called every time someone tried to invoke a nonexistent method on you, then you could very easily implement your own delegation strategy. It would be far easier to code, far more resilient to interface changes, and it would even allow you to abstract your delegation policy into another class, so you could share it with other classes.

Because no syntax changes are needed, the third problem illustrates a class of problems that can be solved using metaprogramming, which lets you change the built-in behavior of classes, e.g. by adding methods or properties, overriding built-in methods, and so on.

Three problem classes, three different techniques: Lexer (or "Reader") macros, evaluator macros, and metaprogramming.

C++ lets you do a *little* of all three of these things with its Template system, and with its operator overloading, which is a limited (but often useful) form of metaprogramming. Not enough with any of them, sadly, and it's too hard to implement what little flexibility it allows you. But it's much better than C's preprocessor, and it's a thousand times better than Java's big fat nothing. It of course would be infinitely better, being a divide-by-zero error, except that we'll give Java some credit for at least not copying C++'s broken templates.

Ruby and Python offer fairly flexible metaprogramming models, but no macros or reader-macros. So they're both susceptible to the first two kinds of problem I mentioned.

Perl has... I dunno. Something. A whole mess of somethings. I know they don't have macros, since they were discussing adding them on the Perl newsgroups a year ago. Perl has some metaprogramming features, but they're relatively limited in scope. And I don't think it has reader macros, although it may offer some sort of preprocessor.

I hope I've demonstrated that reader macros, compiler macros and metaprogramming really can make your code a lot smaller, a lot faster, and a lot more robust. Like any other language feature, you can

abuse them horribly. Unlike other language features, however, you can use macros and metaprogramming to fix broken language features, and in fact make the other features less easily abused.

No language is perfect. The perfect language doesn't exist. A language perfect for one domain can be awful for another. A language that's pretty good today can be awful tomorrow. Because languages aren't perfect, they need to provide mechanisms to let you evolve them to suit your needs. A language's extensibility is one of the most critical keys to its long-term survival.

Again, Java programmers wouldn't be using XML-based build systems, weird HTML/Java templating systems, code generators and all those zillions of other frameworks and tools out there, if Java had been capable of adapting to meet the needs of those users.

Why don't most language implementers add macros and metaprogramming? They *know* that ultimately their language will face extinction if the users can't evolve it. So what are they thinking?

Sometimes they say that it's to protect the users, or make the language friendlier to beginners. That's almost always a baldfaced lie, because they then proceed to pile on horribly confusing features for "experts only". In a few rare cases (Python, Basic and Cobol come to mind), they may actually mean it.

Most of the time, though, it's because they've made it way too hard to implement. The language designer tries really hard to guess which features you'd like, and they create a nice big abstract machine, and a bunch of syntax rules (and parsers for those rules), and semantic rules for interpreting the syntax. After they've piled all that stuff on, their interpreter and/or compiler becomes horribly complex, and the language spec is horribly inconsistent, and they spend all of their time trying to think of ways to fix their busted language.

I tell you: they'd add extensibility if they could. But extensibility has to be designed in from the ground up, and it makes your system many times harder to build. As if designing a language isn't hard enough already.

Wasn't this blog supposed to be about Lisp?

Yup. And now I *think* I'm finally in a position to explain why Lisp is the king of evolution among programming languages.

In stark contrast with every other language out there, Lisp only has two syntactic forms, atoms (i.e. symbols, numbers, strings) and the s-expression (which stands for "symbol expression"), which looks like this:

```
(I am an s-expression.)
```

They can be nested to arbitrary depth:

```
(I am an s-expression.
 (I am a sub-expression.
 (We make a tree, actually.)
 (Pleased to meet you!)))
```

OK, that's Lisp's syntax. Most versions of Lisp add in a small handful of shortcuts, as long as they don't change the overall tree-structure of the code.

And Lisp's runtime machine only needs to support a handful of things:

1. anonymous functions, also called "lambda" expressions
2. named variables
3. nested scopes, one per lambda expression
4. a single control-flow construct called a "continuation"
5. some kind of I/O

Lisp runtimes provide far more than this, of course, but what I've described is the core of Lisp, and it's all you need. In theory, you don't even need special support for strings or numbers; they can be represented by chains of (and compositions of) lambda functions. All operators and control-flow constructs, including conditional logic, loops, exception handling, multithreading, preemptive multitasking, object-oriented programming, *everything* can be implemented using only the five constructs above, and all using the world's simplest syntax.

Try describing the "core of Perl" in anything under a thousand pages — and you still won't be successful.

The system above gives you more metaprogramming ability than any other language, but just in case that's not enough power for you, Lisp has macros: both reader macros and compiler macros.

With reader macros, there's an API to hook into the reader's token stream and make any changes you like before handing the result back to the reader. You can implement preprocessors this way — not just *do* preprocessor stuff, but actually implement arbitrary preprocessors for other people to use. You can change the language syntax however you like: remove all the parens and use whitespace for indentation, for instance, if that's what's most appropriate for your problem domain.

With compiler macros, you can pretty much change anything you like about the language. This scares a lot of people. They prefer their current, very real pain to some imagined possibility of a different kind of pain. Weird, but all too common. Me, I'll take the flexibility of macros, only hire great people, and make sure we all write good code. Problem solved.

I've learned how to use Lisp and Scheme macros, or at least I'm getting a good feel for them, and they're a LOT easier (and more powerful) than C++ templates. Scheme macros use pattern-matching and template substitution. It's similar to XSLT and isn't much harder to learn than XSLT. Lisp macros are just pure Lisp, and are very easy to learn, although it can take a long time to fully appreciate them.

The result: Lisp is pure DNA. You can build *anything* with it, and you always have the ability to evolve even large existing systems in new directions. And that's exactly what you need, because your problem domain is unique.

But I don't like all those parens!

I know. I didn't either. I only started getting used to them maybe a few months ago, and I only started to prefer Lisp to other languages a few days ago. Don't blame ya.

But with what you know now, it should be clear that Lisp's syntax is a technical advantage.

For starters, macros need to parse and generate code, right? That's why C++ templates are so dang complicated. There are a zillion edge-cases to worry about, and that's just in the *syntax*; there are also lots of ill-specified semantic problems with them. If you have a complex syntax, then it's hard for you to implement macros, and it's hard for people to use them. Most languages, in trying to give you lots of syntactic options, have actually limited you permanently to using only those options.

And Lisp is tree-structured code (and data). It's just lists of lists of lists... it's way simpler syntactically

than XML, and we all love XML, right? Well, most people evidently do. Even XML, with its allegedly "ultra-simple" syntax, is still kinda complicated, and working with SAX and DOM parsers isn't entirely trivial. But it beats working with C++/Java/Perl/Python parsers hands-down. Trees are just data structures. We know how to deal with data structures programmatically. But most languages need to have their syntax converted tortuously into a tree structure in order to operate on it programmatically, and by then it's far less recognizable than the original language.

With half a day of work, you could implement XPath expressions to operate on your Lisp code, and start building your own refactoring and analysis tools, rather than waiting around for months, hoping someone else will build them for your Java IDE. In fact, traversing Lisp code is simple enough to do almost trivially in *any* language — but it's far easier in Lisp. So writing Lisp macros and evolving the language to suit your needs is so natural that you wind up doing it all the time, almost from the beginning of a project.

The upshot is that your language gradually winds up being tailored precisely to the system you're building. That's why Emacs-Lisp looks so different from Common Lisp. It's not so different, really, and it supports many features of Common Lisp, even though they differ in a few core areas (as elisp is older, and RMS doesn't care for CL). But Emacs-Lisp has tons of specialized language features designed specifically for editors.

In fact, although I don't have firsthand evidence of this yet, I suspect that the size of Lisp systems tends to grow logarithmically with the size of the feature set. Or maybe an nth-root function. But definitely less than linearly, because unlike in Java, where refactoring tends to make the code base larger, refactoring Lisp code makes it smaller. Lisp has far more abstraction power than Java.

I noticed ten years ago that even though Perl seems superficially concise and expressive, the reality is that adding more features to a Perl system makes the code base grow (roughly) linearly. Need a feature, add a function. Need a system, add a module. It never seems to get easier as your system grows. And Java is even worse; the code grows at a (slightly) greater-than-linear rate as a function of the feature set, because unanticipated design changes can't be bridged with programming or macros, and you wind up having to build the bridges using huge frameworks of objects and interfaces.

So Lisp may seem like it's no better than Perl or Java for small programs — possibly worse, depending on the program. But that's missing the point: Lisp only shines when you use it to build a system that would have been huge in another language. Every sufficiently large problem domain (from JMX to ordering coffee at Starbucks) is best expressed using custom mini-languages, and Lisp lets you build them easily and naturally.

OK, fine. But I don't want to implement a whole language.

Just because you *can* implement specialized language extensions in Lisp doesn't mean that you need to. Common Lisp and Scheme implementations have huge libraries of carefully-chosen, powerful features. Many of these features are unavailable in Java and C++, and some can be expressed in those languages only indirectly.

Hence, ironically, Java and C++ programmers wind up implementing new languages, very awkwardly, using "design patterns" and other heavy frameworks, layering on classes and interfaces in an attempt to build a machine powerful enough to emulate a subset of Common Lisp.

As a language platform, Common Lisp is on par with C++ or Java. It's the *only* language I've found, out of the 20 or 30 that I investigated, that I'd consider to be "production quality". Even Erlang, which is

very mature, still worries me a bit. And Common Lisp is Lisp, which means it will gradually change its shape to fit your problem space in the most natural way.

Common Lisp and many Scheme implementations sport powerful compilers, state-of-the-art garbage collectors, rich IDEs, documentation generators, profilers, and all the other stuff you've come to expect of production-quality languages. They have powerful type systems with the ability to declare static types as desired, to improve performance or readability. There are multiple commercial and free/open-source implementations for both languages. Common Lisp is production-quality and has been for at least as long as C++; Scheme is less ready for prime-time, although it's gradually getting there.

If I were using Common Lisp, I'd definitely miss one or two features from Java, and I'd have to take a few days to implement them.

After all that, I still don't believe you. You're weird.

I know. Don't sweat it. It's just a blog.

I disagree with the previous caller. I want to use Common Lisp at Amazon!

In a word: No.

Languages need large communities *on site* to do well. And you shouldn't underestimate the difficulty of learning Lisp. It generally takes longer to master than even C++ or Perl. Well, about as long, anyway.

We do have a fair number of experienced Lispers scattered about Amazon — maybe thirty or so, more if you count Emacs-Lisp hackers. But you'd really need a bunch of them in *your* group, not just scattered around, in order to get the right critical mass on your team.

And even then, it would be a highly questionable decision. People would be watching you constantly, waiting for you to fail, because most people don't like Lisp and they don't want it to succeed. I was one of those people not too long ago. I was looking for the perfect language, and I never suspected the best candidate would be the one that looks like oatmeal with fingernail clippings. I know *exactly* how Lisp-dislikers feel, and I don't really blame them.

And when you failed, e.g. your service had some sort of highly public outage (as ALL services do, but that would be overlooked), Common Lisp would go on trial, and would be ceremoniously burned at the stake, as a warning to any other would-be Lispers out there. It has happened at many other companies (Yahoo, NASA and JPL come to mind, but there are many of them), and it would happen here. If you somehow managed to get a team of good Lisp hackers together, and you somehow hid the fact that you're using it, and you were lucky enough to have C++ neighbors who have all the outages, then you might get away with it.

But if you didn't get away with it, Lispers would hate you for giving it a bad (well, worse) name. Languages are religions, political campaigns, and social communities, all rolled into one. Never underestimate the ability of one language community to gang up and kill another one. Java, Perl and C++ only thrived here after well over a hundred people per language more or less simultaneously just started using them, and screw you if you think they're not doing their job.

Incidentally, none of our core three languages were allowed at Amazon for the first few years. The only languages allowed were C and Lisp. Lisp was mostly used for huge, much-loved customer-service application called Mailman. Mailman was later hastily rewritten as a somewhat less-loved customer-service application: one that didn't have quite as much mail-composition functionality, but at least it was in Java, which solved the embarrassing problem that nobody in CS Apps at the time knew much

Lisp.

The point is that C++, Perl and Java didn't just show up, they barged right through the front door: first Perl, then C++, then Java, and it took a LONG time before Java folks were viewed as first-class citizens. It's taken them even longer to try to integrate with the existing systems, e.g. the build systems, which were written mostly to solve horrible build/link problems with C++ that don't really exist in Java.

You really don't want to go there. One individual recently tried doing a highly visible internal project in Ruby, and it barfed in a highly public way on its first day of operation; now the Ruby community is mad because it makes Ruby look bad, when Ruby had nothing to do with it. It was a poor decision on the engineer's part. Not because there's anything wrong with Ruby, but because we don't have enough experience with it at Amazon to understand its idioms and performance characteristics, and also because very few people have much experience with it here.

Languages need to "bake" for a few years before they become ready for prime-time at Amazon. They need to be available for personal scripting and productivity tools, and eventually people will start dabbling with getting their feet wet with small systems. Over time, as the systems prove themselves, larger ones are built, and eventually the painful transition to "trusted, first-class language" completes, typically about five years after its first introduction.

Today, there are only three languages that have been well-tested enough in our environment to be considered stable: C, Java, and Perl. And I suppose some people use C++, even though "stable" isn't a very good word for the systems produced with it. For general-purpose programming, and particularly for services that other teams rely on, those are the only languages *I* would use. If a hundred people stood up and announced they were going to start using Common Lisp, I'd probably stand up with them — after asking around to see how well they actually knew it.

But it ain't gonna happen. So don't get too carried away. This is just a speculative blog. Lisp is a cool language, or at least it appears to have great potential. It's worth a look, and maybe even a second look. But don't touch, just look! That Ruby incident has made all the language enthusiasts a bit paranoid, and we're going to be a little extra jumpy for a while.