```
Note, 2021-11-10: I wrote this toward the end of my first year of grad school, in spring 2002,
while trying to find or create a project to work on. Some of my views have changed since then,
but many have not. I had long forgotten writing this until I discovered, through a fortuitous
Google search, a copy that Norman Ramsey archived long ago. Thanks to him for that.

Twenty years on, I am struck by how much of this vision aligns with what Go made real.  -rsc
```

Toward a new systems programming environment

Russ Cox
rsc@mit.edu

1. Introduction

I am growing increasingly dissatisfied with the systems programming environments available to me. This is a rant. In it, I try to articulate what I find wrong with current systems programming environments, in an attempt to decide what the systems programming environment I want to use would look like. Some sections are more well thought out (and thus longer) than others.

To start, I must define systems programming environment. First, systems programming is building programs that provide basic day-to-day system functionality. The line between systems programs and other programs is necessarily fuzzy, but a few examples should suffice. Examples of systems programs include operating system kernels, file systems, window systems, compilers, and editors. Examples of non-systems programs include mathematical simulations, logic programs, theorem provers, AI, and other specialized applications. One approximate characterization might be that systems programs are not difficult algorithmically while non-systems programs often are. That is, systems programs may be well characterized by the fact that their value often lies in their integration and cooperation with other programs rather than in their standalone behavior.

A systems programming environment comprises the various tools used for systems programming. These include programming languages, compilers, editing systems, debuggers, and the operating system itself. That is, C alone is not a systems programming environment at all. C, Unix, Emacs, gcc, and gdb together are a systems programming environment. This distinction is just as important for running programs as for writing them. For example, consider a C program that reads from `/dev/random`. The fact that this file is supposed to return random byte sequences when read is as much a part of the programming environment as C's object model. When such a program is transported to Windows, accomodations must be made for the lack of a `/dev/random` file. In fact, the very existence of files is an abstraction provided by the operating system. There is nothing in the C language that directly supports such a concept.[1] Of course, so-called portable or operating system-independent programs are really still operating system-dependent but only depend on aspects common to all the operating systems of interest. For example, a C program that depends on file I/O is typically considered portable even though it wouldn't run on file-less operating systems like PalmOS. Finally, note that although systems programming environments are tied to various aspects of their operating system, one operating system may support many different systems programming environments. For example, Unix and its default set of shell commands are also a systems programming environment, albeit a much different one from the C-based environment.

[1] This fact is one of the reasons that the ANSI C standard is so complicated. They attempt to remove the dependence on the operating system part of the systems programming environment by growing the language (actually, its libraries) instead. This fact is also at the root of one of the most important differences between Java and Inferno, namely that Java does not define a virtual operating system while Inferno does.

I have just argued at length that the operating system is an inseparable part of the systems programming environment, mainly because that fact may not at first be obvious. Editing systems and debuggers are just as essential to the systems programming environment, although this fact is more well-understood and I will not belabor it further. The important point here is that if we want to build a better systems programming environment, we may need to change much more than just the programming language.

The rest of this rant discusses various axes on which we might measure the suitability of a systems programming environment, at the same time applying these measures to various current or past (sometimes non-systems) programming environments. I have attempted to group the axes somewhat thematically; readers should not infer any sort of precedence from the ordering of the following sections.

Most of these topics are probably well understood in their own subfields or in the context of various languages, and I admit to being fairly ignorant of the relevant literature. There is probably not a single original idea in this rant, but I do believe the synthesis and purpose is new. I list paper references that I have found but not yet explored. Tips about other places to look are appreciated.

2. Interactivity

The interactivity of a programming environment has more to do with ease of writing and debugging programs than most people seem to realize. I assert that interactivity is one of the most important features of a programming environment.

For example, consider your favorite C programming environment. Much is made of features like syntax highlighting, parenthesis or brace matching, auto-indentation, and ctags/cscope-style identifier searching. All current C environments suffer from a much larger problem, namely a lack of interactivity. You can't sit at a prompt, enter C fragments, and see what happens. Such interaction is simply assumed in environments for languages such as Lisp. Interaction facilitates experimentation and provides a convenient way to test individual modules of a program without writing tedious scaffolds. Although it is an imprecise characterization, I will refer to languages typically used without interactive environments (like C, C++, and Java) as non-interactive languages, and to languages typically used with interactive environments (like Lisp, the Unix shell, Smalltalk, Oberon, Python, and ML) as interactive languages.

Ease of experimentation is a very important feature in a language that wants to attract users beyond the designers. Consider the following. How often do you type arbitrary expressions into a Unix shell or other interactive environment just to see what happens, test a hunch, or try out a particularly complex construct before embedding it in a larger program? How often do you write throwaway ten- or twenty-line programs to test similar hunches for non-interactive languages like C.

Ease of testing follows directly from ease of experimentation. It is easier to write and test programs in interactive languages than non-interactive ones. In contrast, programmers writing in non-interactive languages often write C code that isn't tested until it is fit into the context in which it will be run, a much more hostile and less useful testing environment. I was guilty of such sins before Lisp and especially Python taught me the error of my ways, yet I still must force myself to write scaffold ‘‘shells’’ that allow me to reach in and run pieces of a recently written module without needing to connect it to a larger program. Writing such scaffolds is needlessly tedious and would not be necessary in an interactive environment.

The final important feature of interactive programming languages is that they reduce the time between writing code and seeing it run, making it easy to run code immediately after it has run and thus build code from the bottom up testing each piece as you go. Draper [draper, ppig96] points out that the event of running a program for the first time carries much more psychological weight than might logically be expected, because that first run is also testing all the latent unconscious decisions present in its design, any one of which might manifest itself as an unanticipated disaster. Thus interactive environments provide a leg up even psychologically.

Of course, it's not entirely fair to categorize languages by the features of their common programming environments. Interactive environments for Pascal and C have been built (cin, Saber-C, PECAN), and there are probably a few Lisp and ML programmers who completely ignore the rich interactivity of their languages' default environments. Still, the interactivity of a language's environment derives in large part from the model presented by the language and the interface presented by the first implementation. Most of the interactive languages have been those built around expressions as the centerpiece of the language, although the existence of interactive Pascal and C environments demonstrates that expression-centricity is not a requirement for having an interactive environment.

need to find:
[Delisle et al., Viewing a Programming Environment as a Single Tool, Software Engineering Symposium on Practical Software Development Environments, April 24, 1984, 49-56.]
[Sandewall, Programming in an Interactive Environment: The ‘‘LISP’’ Experience. Computing Surveys 10, 1 (March, 1978), 35-71.]
[Kaufer et al., Saber-C: An Interpreter-based Programming Environment for the C Language. Usenix 1988 Summer Conference Proceedings, 161-171.]
[Feuer, si: An Interpreter for the C language. Usenix 1985 Summer Conference Proceedings, 47-55.]

3. Language extension

One feature we might want in a programming language is that it be easily extended. Lisp was the first language to get this right, and is still the clear winner, in more ways than one.

To start, it would be nice if we could extend a language to produce interpreters for specialized ‘‘little languages’’ akin to Yacc or Burg but without needing to write mundane things like parsers. One way to do this is to express these ‘‘little programs’’ as some sort of complex data structure in the parent language. Lisp's nested lists (S expressions) were the first such complex data structure, of course, but any language with decent nested list or tuple support would suffice. A yacc-style parser description could easily be written as an S expression of rules and actions in Lisp, Python, or ML. What distinguishes Lisp is that such a parser description wouldn't look much different from a normal Lisp program: Lisp programmers already express their programs as S-expressions, while Python and ML programmers typically do not. Put another way, Lisp avoids having a difficult barrier between programs and data by putting the difficult barrier *before* the programs: you have to leap the barrier in order to write *anything*, but once you've leapt it, writing data as programs doesn't require any more work. That is, Lisp is the clear winner because little programs in Lisp don't look any different than big ones.

Lisp takes this much farther, though, in ways that are not immediately obvious to seasoned C hackers like me. Since Lisp programs have no syntax other than the S expressions, the language is easily extended. As proof by contrast, suppose we want to define a new operator ** to perform exponentiation, in analogue to the multiplication operator *. In C, making the analogy as true as possible would require changing the parser on the fly to admit ** as a new infix operator. If we were willing to settle for a prefix operator, we still wouldn't be able to spell it **; allowing such names would introduce parsing ambiguities with the infix notation. On the other hand, Lisp, content with its prefix operator syntax, requires no such changes. As another example, suppose we wanted to overload * to allow multiplication of strings. In C, this is impossible; C++ makes it possible, but only for the special case of extant operators: defining ** for strings would still be impossible. In Lisp, defining multiplication on strings is only slightly more complicated than defining `**` was (because the implementation must check the type of its operands). It is worth pointing out that ML addresses these concerns by using a parser that can modify itself on demand to accomodate such changes. Although useful and syntactically prettier, this approach is not without its own occasional surprises.

Finally, Lisp extensions need not be normal functions. Through its macro facilities, Lisp exposes the building blocks necessary to build constructs like `if` and `while`. Once armed with this power, it is easy to add other control structures (for example, short-circuit booleans) to the language. This frees the language designer from anticipating all the necessary control structures. If the programmer wants a new control structure, he can code it himself, *in Lisp*, without changing the compiler. Other languages have adopted this trick as well. For example, Smalltalk and Tcl both expose ‘‘blocks,’’ which suffice to implement arbitrary control structures, among other things.

In short, common systems programming languages like C, C++, and Java can’t hold a candle to Lisp as far as extensibility. They can’t define complex constant S-expressions and don’t allow the programmer to introduce new control structures as needed.

It is only slightly unfair to observe that Lisp provides this great extensibility at the cost of providing what some would consider significantly degraded syntax. This seems like a small price to pay, but good clean syntax encourages program readability, which in turn is quite valuable. It’s a difficult trade to consider. Perhaps there is some other way to make programs and data indistinguishable, but that seems doubtful. Paul Graham’s design of Arc, a new Lisp, suggests that one way to address this is to have S-expressions underneath but have syntactic sugar on top. If you do this, though, it seems to reintroduce the syntactic barrier between programs and data, and part of the reason that Lisp is so good at extension is the lack of any barrier there.

4. Playing well with others

Programming environments differ wildly in how easy it is to compose programs and to reuse code.

4.1. Coarse-grained program composition

By program composition I mean connecting two programs in some useful way. The Unix shell is the programming environment that has done the most to promote such composition. In a 1964 memo, Doug McIlroy toyed with the ideas that would become Unix pipelines: ‘‘We should have some ways of coupling programs like garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.’’ What made Unix popular was the ease with which programs could be connected in linear pipeline fashion. Of course, the interface that programs must meet is a strict one: reading a stream from standard input and writing a stream to standard output. There can be no backward flow of data, no real interaction between programs. Still, this style is quite powerful and lives on today in many useful guises.

At the same time, the Unix shell style of program composition isn’t sufficient when you want two programs to communicate in both directions or want complicated patterns of communication. In Unix, the traditional workaround is to abandon the shell programming environment in favor of C. C programs are still shell programs, in the sense that they can be run from shell programs, but they’re definitely different beasts. The important difference is that running another program from within a C program requires fiddling with lots of details that the shell hides. In return for fiddling with these details, the programmer gets more control over the interaction with the other program. A happy middle ground might be a way to run other programs that had sensible shell-like defaults that could be easily overridden without needing to specify all the details required by C. Notice that the clients of the interactive programs are suffering the burden of connection infrastructure. Every program that wants to interact with a popular other program must duplicate the same connection infrastructure to gain interactivity. It is interesting to note that Plan 9 addresses this problem by encouraging such popular programs to be file servers. Interaction then becomes a simple matter of reading and writing files, a task shell scripts are adept at.

Difficulties of connecting programs aside, both Unix and Plan 9 have the property that the interface between programs is restricted to streams of bytes. This is a reasonable low-level abstraction, but the nature of C and shell programs often prevents general higher-level abstractions from being built on top of it. There's no general way to take an arbitrary C data structure and pass it to another C program, because there is no general way to marshal that data structure into a stream of bytes and back. Conventionally, Unix and Plan 9 programs use text as the formatting for their byte streams, making conversion even more difficult. Using text has the benefit of making programs easier for humans to interact with, a win for interactivity as discussed earlier. Still, there are times when it would be reasonable to forego text if that meant an easier time for the programmer. If there were a general way to format data into byte streams, then a general pair of tools could convert between the binary representation and human-readable text when necessary.

There are really two problems here: the first is that the interface between programs is too constrained for some purposes, and the second is that the language does not have adequate tools for coping with that constraint. Operating systems like the Exokernel and SPIN showed the benefits of addressing the first problem; those systems tore down the rigid interface between kernel and user programs, replacing it with a more flexible function call interface. There are times (for example, when communicating over a network) when the first problem is unavoidable. Languages like Java and Python provide support for general marshalling routines that do effectively address the second problem.

4.2. Fine-grained program composition

The discussion to this point has concentrated on coarse-grained program composition, in which one program is typically useful by itself without the other. By fine-grained program composition I mean putting programs together in ways commonly associated with code libraries, when you use a routine from here, another routine from there.

The reason there are so many packages for Python and Perl is that it's trivial in those languages to reuse smaller library pieces, and that's just not true of Plan 9 C or Alef. It *is* true of shell scripting, but then you're stuck with the constrained byte stream interface for screwing programs together.

Wirth makes a very good argument in his compiler construction book that dynamic linking is really the only way to go, not dynamic linking as in C, but dynamic loading and linking as in Limbo or Java or the Unix shell. The main argument for dynamic linking is that when one component of a program changes, as long as the interface remains the same, any components that interact with the changed components can be left alone. Think how painful it would be to use Unix if every shell script that called `sort` or `awk` had to be revisited when either of those programs was changed.

At a finer level, I have the most experience using dynamic linking in Limbo, where modules are linked in at run time through the `load` construct in the language. It is very nice to be able to change common library routines (say, to fix a bug) without recompiling all the programs that use that library. I think this is definitely a good feature in a programming environment. It makes building a standalone program more difficult, since the set of modules that program loads is only determined at run-time and might even be input-dependent. This is slightly unfortunate but also very powerful. I think this is the right way to do libraries.

4.3. Operating systems

In my mind, operating systems serve three purposes. The first is to provide an abstraction for various aspects of the hardware. The second is to insulate programs from each other, and the third is to help programs to communicate with each other.

The first purpose, providing an abstraction for the hardware, will always be with us, and I'm not going to consider it further.

The Oberon system is peculiar in that the second two purposes aren't really served by the OS. Instead, the dynamic loading and linking in the language is used to compose programs, and there is little to no protection between programs other than the safety provided by the language and compiler.

I think the arbitrary composition achieved by Oberon (and, to a lesser extent, SPIN and the Exokernel) is a very powerful mechanism and is worth considering as a replacement for the usual operating system process communication primitives. Doing so requires that everything runs in the same address space, which complicates things like garbage collection, process termination, and sharing a machine between many users. I think a mix must be the way to go, but the details of the mix aren't clear.

Let's suppose our operating system provides some Interprocess object sharing mechanism. It's not clear how to handle access controls. The nice thing about the file system abstractions of Plan 9 is that they have a clear access moderation policy. Perhaps it would make sense to make the object sharing mechanism publish objects in a named hierarchy with Unix-like permissions.

Another problem is how and when to ship code around, if at all.

5. Types

I'm not at all clear on what types are really necessary in a good systems language. I think we'd get a lot of leverage out of a dynamically typed system with any amount of static typing on the side.

5.1. Reflection

Whatever the type system, reflection solves so many problems that I think it is basically essential. If we have reflection we can do things like marshalling or SUN RPC without any extra help.

One interesting possibility is to compile the marshallers on-the-fly to get the speed of compiled code. I think this would work best if the language is compiling to byte codes. Then one compiler works everywhere, and the byte codes get jitted as part of normal running. That could be neat, but is a separate issue.

5.2. Polymorphism

Polymorphism is very powerful and very convenient but it's not clear how to put it into the system. Dynamic typing would get you most of the same safety, and then the compiler could do some checking but not need any implementation underneath. I've not thought enough about this.

6. Concurrency

The world runs in parallel, but our programming languages don't. In my opinion, a good systems programming language must support dealing with this concurrency. The best way to do this (again, in my opinion) is to make very lightweight (probably coroutine-based) threading easy, in the style of ‘‘Bell Labs CSP’’ as discussed in my PDOS threading talk or Harvard's CS257 concurrency seminar.

The self-certifying file system SFS is built on a C++ library called `libasync`, which provides a mechanism for manually writing programs in the continuation-passing style [Appel]. Programming in this style requires sacrificing a lot of notational and syntactic clarity. Even constructs like `for` loops must be shunned if anything in the body might want to relinquish the processor to another locus of execution.

A lightweight threading library like Plan 9's `libthread` is another way to achieve the same results, but without sacrificing the notational clarity. The problem with `libthread` is that stacks are allocated statically, so that you must either waste space or worry about stack overflow (or both!). This is typically discounted in the Plan 9 model, where we don't commonly invoke stack-heavy functions and have few enough threads that having eight kilobyte stacks doesn't hurt us. Still, our idle threads are typically sitting on less than 300 bytes of stack, nowhere near the eight kilobytes.

There are two responses to this stack problem. One is that SFS has relatively few threads anyway, so stack overallocation would not be noticeably wasteful. Of course, we might want to consider network servers with many threads. For example, modern IRC servers typically handle about 20,000 simultaneous connections. A better idea is to coerce the compiler into allocating pieces of stack incrementally, much as is done by Limbo/Inferno. Then we avoid worrying about overflow without overprovisioning.

Once we have good thread support, it might be possible to build a kernel with similarly good thread support, replacing `select` with many threads executing the desired I/O operations. One benefit of this is that disk I/O would then be overlapped with other work, whereas now disk I/O just stops `libasync` programs, since on-disk files cannot be `select`ed.

It is worth pointing out that the model of threading I am proposing is not the typical ‘‘Birrell-style’’ threading taught in operating systems or Java classes. Libasync is easy to use conceptually (once you get past the awful continuations) because there is no actual parallelism, just interlacing at well-defined points. Thus there is no need for locking to protect most access of shared structures. The `libthread` approach would have the same benefits, but avoid the continuation syntax problems.

Some have raised concerns about the fact that in `libthread` you can't tell whether a particular function might yield the processor, whereas in `libasync` it is obvious: a function can only yield the processor if you pass it a return continuation. I don't have an argument here except that in many years of writing threaded programs in this model under Plan 9 I've never had any problems wondering whether a given function call might yield to another thread. I think most of this has to do with the granularity of most necessarily-atomic operations. Typical operations that must be performed atomically are a few statements in a row (for example, manipulating pointers in a doubly-linked list) rather than things that include calls to arbitrary functions. The ‘‘obvious scheduling’’ issue is one that I think can only be resolved by building a system and seeing if people get confused.

7. Garbage collection

Garbage collection is perhaps the most religious issue I bring up in this rant. Systems programmers have long shunned garbage collection; its association with the functional programming community is at least partly responsible for its being found ritually impure.

7.1. Semantics

Leaving aside implementation for the moment, there are good arguments for and against having garbage collection in a language. Most user-level systems programs would be cleaner if relieved of the burden of automatic storage reclamation. On the other hand, there are parts of a kernel that need more control over pointers than a garbage-collected language affords. The obvious conclusion to this is that a good systems programming language should provide both choices: garbage collection *and* manual storage management. This is not a new observation: Modula-2+ balanced both gracefully by having separate type spaces and distinguishing modules as safe (garbage collection only) or unsafe (uses pointers).[2]

7.2. Efficiency

The most common knee jerk argument against garbage collection is that it is too inefficient, either in space or in time. Of course, the details depend on the particular strategy chosen, but we can make some general observations.

First, most malloc implementations also fall somewhere along the fast-small continuum, and not many people seem to care. At one ends, a simple free list scheme with fixed size blocks can be quite fast although not particularly space efficient and might be subject to internal heap fragmentation problems. At the other end, a tree-based allocator like the Plan 9 allocator is relatively slow but takes great pains to be space efficient and avoid heap fragmentation whenever possible. Most users of fast space-inefficient mallocs seem not to be affected too much by the waste of space. Similarly, in four years of using the current Plan 9 allocator we have never encountered a cpu-bound program whose bottleneck was the allocator.

Second, many state-of-the-art garbage collectors are competitive with malloc/free implementations in speed and space requirements. (Admittedly, copying collectors have that pesky factor of two, but there are good state-of-the-art mark and sweep collectors as well.)

Finally, machine efficiency is often irrelevant. As long as the asymptotics are right, the small constant factors are greatly outweighed by the increase in *programmer efficiency* that comes from not worrying about storage reclamation and not tracking down pointer bugs. Another of the reasons the Plan 9 allocator can be so slow is that it tries to identify poorly-behaving clients early. In this respect it functions as a poor man's Purify. I have spent entire days using it to track down memory leaks and dangling pointer uses that would have taken much longer to track down with a less helpful allocator. I would like those days of my life back, but I would settle for working toward a way to avoid losing many more.

7.3. Concurrency

Remember when I said modern garbage collectors were competitive with modern malloc/free implementations? I lied. While the statement is true for single-threaded programs, the sad fact is that the garbage collection community doesn't have a clue how to do collect programs with multiple threads generally, simply, and efficiently. Collectors that require global knowledge (like mark-and-sweep and copying collectors) must either stop all the threads while they work or must have some method of discovering what the threads are doing (specifically, how they are modifying pointer cells) during the collection. Software-based discovery mechanisms typically involve turning pointer assignment into a much more expensive operation than a simple memory instruction. Hardware-based discovery mechanisms use virtual memory protection to allow pointer-ignorant threads to execute, but those that trip over the memory protected pages typically stop until collection finishes. Modula-2+ had one of the more elegant software-based solutions, using a transaction queue to log pointer operations and send them *en masse* to the garbage collection thread. This is still quite expensive, though.

On the other end of the spectrum, reference counting does not require stopping all the threads simultaneously, but is not general: it cannot reclaim cyclic data. It does make pointer assignment more expensive, but often not as expensive as the software discovery mechanisms

[2] The distinction between safe and unsafe goes farther than this criteria, but is not relevant here.

discussed above. Modula-2+ used reference counting to collect most data, combined with much more infrequent mark-and-sweep passes informed by the transaction queue mentioned above. Reference counting also has the advantage of being much simpler than any of the above mechanisms, and the memory it does reclaim can be reclaimed at the instant it becomes available.

Depending on the concurrency model, stopping all the threads for a global knowledge collector may not be too expensive or too noticeable. If threads are cooperatively scheduled, it is easy to stop them for a collector, and they can arrange to stop only at safe points. On the other hand, if threads are preemptively scheduled or run in parallel (on different processors), it becomes more expensive and more obvious when they all stop.

Also, reference counting may not be as expensive as it is reputed to be. First, studies of languages like Smalltalk and Modula-2+ have shown that most objects have exactly one reference and aren't involved in cyclic data structures. Thus reference counting, although not completely general, is sufficient for large classes of data. It is possible to identify non-cyclic objects in the type system, as was done in Limbo/Inferno with programmer annotations. Those objects can then be kept separate from data that must be collected using global knowledge. It might be useful to restrict oneself to such types when writing the low levels of an operating system kernel, for example.

Further, since most objects have only one referent, perhaps it is possible to identify those at compile time, completely avoiding reference counting overhead for those. Jones and Lins mention a few papers about linear logic and Clean's unique types, [girard, 1987] [brus et al. 1987] but they are quite theoretical and I have not examined them closely. Jones and Lins mention a number of papers about doing reference counting inside the compiler (rather than at run-time), [hudak, 1986] [brus et al 1987] [cann and oldehoeft, 1988] [hederman, 1988] [baker, 1994] but none of them are online.

Finally, systems programmers have been using manual reference counting for a long time with minimal overhead, because they only change reference counts when absolutely necessary rather than at every assignment. Perhaps it is possible to move these intuitions into a compiler to get cheap reference counting.

Regions [bobrow, 1980] [aiken and gay, 2001] are a promising way to make reference counts work for cyclic data, by arranging to count only pointers from outside the cycle. They require more programmer annotation than I am comfortable with, and currently no one knows how to make compilers do the annotating themselves.

The conclusion is that I really don't know what to do here. I want garbage collection in my concurrent programs, but there isn't a clear way to implement it. Global knowledge collectors need to stop all the threads, which may or may not be okay. Reference counting avoids global knowledge and potentially long pause times, but isn't completely general and would thus need to be supplemented by a global knowledge collector or by requiring programmers to break loops manually. A supplemental collector could run less frequently, reducing its expense. I'm leaning toward believing the right answer is supplemented reference counts rather than depend on a global knowledge collector for everything.

8. Acme

I think there's a lot to learn from the Plan 9 editor/window system/shell hybrid Acme, but I don't have time to expand on it now.

9. Not invented here

The obvious question, having been through all this, is ‘‘why not just use Lisp, or Modula-3, or ...?’’ In an essay entitled ‘‘The Roots of Lisp,’’ Paul Graham observes:

> It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new langauges being developed have been moving steadily toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts from the Lisp model, like runtime typing and garbage collection.

Am I just contributing to the swampy lowlands? I don’t really know. I hope not. I hesitate to embrace all of Lisp because of the compilation and run-time difficulties that doing so would entail. I think that I can get the features I want at reasonable performance with less implementation effort than starting with Lisp. One of the Smalltalk guys said that Lisp isn’t a programming language: it’s building material for programming languages. That’s the main reason I shy away from it. I don’t need that much.

Here I’ll try to point out where I think each language comes up short. Whether I can do any better is, of course, a matter not yet settled.

9.1. C

I’ve been complaining about C this whole document. It should be clear what I miss in using C.

9.2. C++

C++ doesn’t go far enough in some places (reflection, concurrency, garbage collection) and is far too complicated in most places. The main problem with C++ is that it is so enormous. Most C++ programmers learn some fraction (let’s say 30%) of the language, but the 30% they use is different from the 30% some other people use, leading to code that can’t be easily shared between people. Also, the C++ compilation model is just broken. Templates are a poor substitute for actual polymorphism, and lead to greatly inflated binaries. It’s not clear that inheriting implementations (as opposed to interfaces) is a good thing at all.

9.3. Java

Java is a nice language. If the implementations weren’t so slow, it might be worth using. It’s got a good object model, good reflection capabilities, and garbage collection. It takes a stab at threading, but the result is the complicated Birrell threads rather than the threading I am proposing. I think the loader might be a bit too magical (it certainly seems that way), and it’s a bit too painful to connection Java programs over a network (with all the RMI stubs, etc.).

9.4. Modula-*n*

I think Modula is a neat language, though I’m put off by the syntax. The Birrell threading model is just wrong as far as I’m concerned, though.

9.5. Oberon

Oberon is a particularly clear language and system, though its syntax is far too verbose for my tastes. I think there’s a lot to learn from Oberon. There’s no threading support, and the language is, to some extent, tailored to the fact that it’s a single-image single-user terminal system. I’d like to pick up the good things but use them in broader contexts.

9.6. Lisp

Paul Graham is designing a new language called Arc,[3] which is basically going to be Lisp fixed in various ways: perhaps there will be more legible syntax, and there will be more access to the innards of the environment. A better question than ‘‘why not just use Lisp?’’ is ‘‘why not just use Arc?’’ Again I think the reason is that I don’t need all that. Sure I can implement concurrency with `call/cc`, but that doesn’t make it a good idea if I’m interested in performance. Using a language like Lisp or Arc abstracts too much and keeps me too far from the underlying implementation of the primitives I want.

9.7. ML

ML is not suitable for writing systems programs. Type inference causes no end of confusion, and most systems tasks lend themselves more easily to iteration and sequential execution rather than functional definitions. ML is just too much of a mind warp. I’m targeting a different audience than ML does.

9.8. Smalltalk

Smalltalk is one of my inspirations for thinking about this. When I play with Squeak (the current Smalltalk implementation), I get the sense that this could be a really neat environment though I don’t understand a lot of it. Smalltalk, like Oberon, demonstrates the benefits of dynamic linking and the simplification sof a single-image single-user terminal system. I’d like to break out of this simplification if possible, and I think Smalltalk syntax is a bit arcane for me. As usual, they don’t have the threading I want.

10. The road ahead

I want to build a *system*. Not a piece, but a system, something where all the pieces fit and work well together. Plan 9 doesn’t cohere the way that I imagine Oberon or Smalltalk did, though this might be a green grass problem: Plan 9 certainly coheres better than Windows, which in turn coheres (in my mind) better than Linux and FreeBSD currently do.

A good incremental step would be to build a compiler for the language and use it to write programs for current operating systems, like Plan 9, FreeBSD, Linux, and Windows. Even this can be done incrementally, starting with a simple language and adding advanced features as needed.

As a first milestone, I propose to reimplement SFS in the new language using threads in place of `libasync`, perhaps using a C/`libthread` implementation as a stepping stone. Comparing the threaded and `libasync` SFS implementations is a good self-contained piece that might, depending on the results, make a decent SOSP submission for next spring.

After that, future milestones seem less clear. It seems likely that they could take the form of comparing implementations (like the SFS comparison) or describing how some feature of the language facilitated a particularly clean way to structure things. As another example, I think it might be interesting to write a multilingual file system protocol translator (9P, NFS, SMB, FTP, SFS) by composing threaded modules. It seems like that would be particularly clean in a language with good object and thread support as opposed to C or C++.

Eventually, I hope there would be enough experience building and connecting programs written in this language to think about what sorts of support would be nice to have in the operating system in a system structured only around this language.

[3] His web site at www.paulgraham.com contains a wealth of information about Arc as well as general information of interest to language designers. It is well worth exploring.

I realize that it’s not clear exactly how my systems programming environment would differ significantly from Oberon or Modula-2+. The particular threading model is a big difference, of course, but not a new idea. It would be interesting to learn how the model pans out when trying to write high-performance system software. I don’t want to dwell too much on performance, though.

My goal is to build a language and more importantly a programming environment that makes systems programming significantly less painful than it currently is, hopefully approaching pleasant. I think there will be interesting things to find down that path, though I don’t know exactly what those will be. My plan is that each feature subsystem will be carefully reasoned and compared with other alternatives, so that at the end of the project, I’ll have not only a good systems environment but an explanation of why it’s better than the alternatives.